UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA


# Imposição de Segurança em Aplicações Web a partir de Linguagem Intermédia

Miguel Carvalho Fernandes e Simões Moreira

**Mestrado em Engenharia Informática**
Especialização em Engenharia de Software

Dissertação orientada por:
Prof.ª Doutora Ibéria Vitória de Sousa Medeiros
Prof. Doutor Francisco Cipriano da Cunha Martins

2021

# Agradecimentos

Em primeiro lugar quero a agradecer ao meu professor e orientador Francisco Martins pela proposta que me fez de uma tese mais desafiante, a qual sem os seus pareceres técnicos nos momentos certos não teria sido possível concretizar. Quero também agradecer à minha orientadora Ibéria Medeiros que me ajudou no decurso desta longa tese, que me fez acreditar que a tese ía num bom caminho nos meus momentos de dúvida, esse caminho só foi possível com as suas orientações. Aproveito para agradecer à minha família que sempre me apoiou ao longo de todo este percurso. Agradeço também a todos os meus colegas da unidade de investigação do LASIGE, em especial ao Diogo Sousa que me mostrou a linguagem de baixo nível utilizada neste trabalho e que me deu pequenas dicas relacionadas com programação em baixo nível. Este trabalho foi parcialmente suportado pelo FCT através do projeto SEAL (PTDC/CCI-INF/29058/2017) e pela unidade de investigação LASIGE (UIDB/00408/2020).

*À minha família e aos meus amigos*

# Resumo

As aplicações e serviços web são atualmente os meios mais comuns de acesso a recursos organizacionais. Linguagens de *scripting* são normalmente as mais utilizadas para desenvolver estas aplicações pelo facto de permitirem um rápido desenvolvimento, devido a terem muitas extensões de bibliotecas, um rápido ciclo de desenvolvimento, tipos dinâmicos e polimorfismo. Devido à sua disponibilidade na Internet para acesso a serviços e recursos, estas ficam expostas e tornam-se um alvo de diversos ataques (ex., injeções de SQL e *Cross-Site Scripting*) que exploram vulnerabilidades presentes no seu código fonte, pondo em risco diversos serviços e gerando grandes perdas e estragos às organizações. As aplicações web cada vez mais integram várias tecnologias e linguagens, fazendo com que o seu código seja mais complexo e consequentemente mais difícil de analisar estaticamente e descobrir vulnerabilidades. Por seu termo, a representação do seu código numa linguagem intermédia, de baixo nível, remove esta complexidade, mas mantem os aspectos fundamentais e a semântica do código.

Esta dissertação tem como objetivo a deteção de vulnerabilidades numa linguagem intermédia, de baixo nível, para aplicações desenvolvidas em PHP, a linguagem mais usada na construção de aplicações web. É utilizada a linguagem de baixo nível HipHop *assembly* (HHAS) para representar o PHP. Esta linguagem foi desenvolvida explicitamente para compilar programas escritos em Hack e em PHP. Foi criado um analisador estático de código capaz de analisar HHAS e de descobrir vulnerabilidades em programas compilados para esta linguagem, sem que seja preciso executar o programa. Este analisador estático recorre a um sistema de tipos e a técnicas de *taint analysis* (análise de comprometimento) para verificar se um programa é ou não vulnerável para todas as possíveis execuções. Os tipos de ataques detetados por essa análise são a injeções de SQL e o *Cross-Site Scripting*, as duas classes mais exploradas em aplicações web. A ferramenta desenvolvida foi avaliada com código PHP, verificando-se que a análise por ela realizada sobre linguagens que exigem rápidos ciclos de desenvolvimento e que permitem a utilização de funcionalidades dinâmicas, tem esses aspectos em consideração.

**Palavras-chave:** vulnerabilidades web, linguagem intermédia de baixo nível, análise estática de código, sistema de tipos, análise de comprometimento.

# Abstract

Society and business rely on software systems, which are integrated into a complex system that depends on other systems (e.g., automation systems, business systems, Internet of things (IoT), and mobile devices). These systems are mostly web applications and web services, which are used as the most common ways to access organizational resources. Such applications and services are developed by scripting languages to perform server-side processing and access to systems resources (e.g., databases).

Scripting languages, like PHP, are usually the most used because they allow rapid development, have many library extensions, a fast development cycle, dynamic types, and polymorphism. Due to its availability on the Internet, web applications developed in such languages are exposed, making them appellative targets for several and diverse attacks (e.g., SQL Injections (SQLI) and Cross-Site Scripting (XSS)) that try to exploit vulnerabilities present in their source code, endangering several services, and generating large losses and damages to companies.

Many of the vulnerabilities exploited by attackers are related to the lack of sanitisation and validation of user inputs, which later will reach sensitive sinks (possible vulnerable areas of the code). Therefore, a good practice for web application security is to validate user inputs with sanitisation functions, i.e., functions that evaluate their content and invalidate dangerous characters.

Programmers often use static analysis tools to automatically look for vulnerabilities in the application's source code. However, developing these tools requires specific knowledge of the code and about each functionality of the language, which is a hard and complex task. There are several approaches to identify and prevent the exploitation of SQLI and XSS vulnerabilities, namely, through the use of defensive code, static analysis, dynamic monitoring, and test generation. While these approaches contribute to the identification of vulnerabilities, they also offer an opportunity for improvement. The approaches that depend on defensive coding are subject to errors and demand the rewritten of existing software with secure libraries. Static analysis tools are more likely to produce false positives. Dynamic monitoring tools increase the run time of applications and to detect vulnerabilities applications must be running. White-box tests are developed taking into account the source

code, unlike black-box tests which have no knowledge of the code, but that allows the discovery of new vulnerabilities.

Also, it is known that the growth of software technologies development is not matched by the growth in security investments. The number of vulnerabilities in web applications has been increasing due to the development of new techniques for computer attacks. Web applications increasingly integrate more technologies and languages, making it more complex to discover vulnerabilities in their code. For these reasons, the representation of its code in a low-level, intermediate language, removes some of this complexity and keeps the fundamental aspects and semantic of the code.

This dissertation has the goal to present a type system and a tool to detect programs vulnerable to SQLI and XSS attacks, in a intermediate language, for applications developed in PHP, the most used language to build web applications. The low-level language HipHop assembly (HHAS) is used to represent PHP applications, allowing the tool to detect vulnerabilities in a simplified way through static analysis.

HHAS is an intermediate language that allows representing PHP and Hack (the language used in the development of Facebook) at the lowest level, maintaining the characteristics of the language at a high level. HHAS is the representation of the HipHop bytecode (HHBC) language consumed by the HipHop virtual machine (HHVM), which is readable and written by humans. Unlike PHP's native statements, which only describe what actions are to be taken, HHAS does not use two different instructions that produce the same result, telling the machine how these actions are to be carried out. In addition, HHAS introduces the concept of *operand stack*, a space in the stack where data is inserted or removed depending on the different operations being processed. The representation in bytecode also minimizes the work done at runtime and allows to make optimizations and analyses in advance.

The dissertation also presents the DVHipHop tool, which implements a static analyzer, based on the type system we defined that is capable of analysing HHAS and discovering vulnerabilities in applications compiled to this language, without having to run the application. The tool uses the type system based on tainted analysis to check whether a program is vulnerable or not to all possible executions. The fact of this static analysis is done in a lower-level language makes it possible that in the future the same security checks can be made for other programs written in other languages.

Attacks such as SQLI and XSS can come in different ways and can reach different kinds of sensitive sinks. This means that, a tainted value (i.e., a malicious value) can be sanitised by a sanitisation function for one sensitive sink but not for other. To develop an analysis that takes into account multiple functionalities of a dynamic language, like PHP, we choose to abstract this fact and consider just one type of

tainted and untainted values (i.e., malicious and non-malicious).

To evaluate the developed tool, we used over 8224 tests of the SARD dataset for SQLI and XSS vulnerabilities. The tool achieved good results and found the 250 vulnerable tests. We also created specific tests to test the tool for its full capacity in the discovery of vulnerabilities in programs that use all functionality considered by the tool. In some tests, the tool just needs to consider one functionality at a time, in others, it needs to consider multiple functionalities in one program, for example, aliases between global variables and function calls. This evaluation allowed us to conclude that the developed solution does what was expected to do. Also, the development of a type system helps to define, in a formal way, our type analyses and consequently prevent our tool to originate false negatives.

The main contributions of this dissertation are the following: (1) a study of the different types of taint analysis and static code analysis; (2) definition of a type analysis, based on a taint analysis, through a type system, which looks at all characteristic of an imperative language and takes into account functionalities like associative arrays, global variables, aliases and variables dynamically accessed; (4) a tool capable of analysing web applications translated to HHAS and identifying vulnerabilities to SQLI and XSS attacks; (5) an evaluation of the tool through two sets of tests.

**Keywords:** web vulnerabilities, low-level intermediate language, static code analysis, type system, taint analysis

# Conteúdo

# Lista de Figuras

# Lista de Listagens

# Lista de Tabelas

# Capítulo 1

# Introdução

As aplicações *web* são normalmente desenvolvidas por linguagens de *script* as quais são utilizadas pelo facto de permitirem um rápido desenvolvimento, por terem muitas extensões, rápidos ciclos de desenvolvimento e tipos dinâmicos. O PHP é uma linguagem deste tipo e é a mais utilizada no desenvolvimento de aplicações do lado do servidor [3]. Estas linguagens dinâmicas são normalmente interpretadas [4] e por isso estão sujeitas a um tempo de execução mais elevado do que o de uma linguagem compilada. Para além disso, os programas escritos nestas linguagens estão muitas vezes vulneráveis a ataques de utilizadores que com elas interagem.

Devido a estas características que facilitam o desenvolvimento destas aplicações, em troca de desempenho e segurança, têm vindo a ser desenvolvidas algumas ferramentas para melhorar o seu desempenho. Por exemplo, o compilador HipHop é um compilador estático desenvolvido para o PHP [4], e o compilador HappyJIT que usa *just-in-time compilation* em código escrito em PHP [5]. Por seu turno, para o desenvolvimento da segurança têm sido desenvolvidas ferramentas que usam diferentes técnicas, para a deteção de vulnerabilidades, tais como as de análise estática de código [6], testes de penetração automática [7] e aprendizagem automática combinada com análise estática [2, 8]. O DEKANT [2] é uma destas ferramentas, que através da aprendizagem automática aprende a detetar vulnerabilidades estaticamente. O WAP [8] é outra ferramenta que recorre a aprendizagem automática para confirmar se as vulnerabilidades detectadas estaticamente são de facto reais.

Muitas das vulnerabilidades exploradas pelos atacantes estão relacionadas com a não sanitização e validação das entradas dos utilizadores (inputs), os quais são usados em zonas do código sensíveis e possivelmente vulneráveis. Portanto, uma boa prática para segurança de aplicações web é validar as entradas de utilizador com funções de sanitização, i.e., funções que avaliam o seu conteúdo e que invalidam conteúdos perigosos. Os programadores frequentemente usam ferramentas de análise estática para procurar vulnerabilidades automaticamente no código fonte das aplicações. No entanto, desenvolver estas ferramentas requer um especial co-

nhecimento do código e sobre cada funcionalidade de uma linguagem, o que é uma tarefa difícil e complexa. Para além disso, caso a ferramenta não tenha em conta uma função que sanitize entradas do utilizador, esta pode gerar falsos positivos [2].

## 1.1 Motivação

A necessidade de garantir a qualidade do *software* nunca foi tão grande. A sociedade e os negócios estão dependentes de sistemas de *software* que estão integrados num complexo sistema que depende de outros (ex., automatização de sistemas, sistemas de negócios, Internet das coisas (IoT) e dispositivos móveis).

Infelizmente, o crescimento do desenvolvimento de tecnologias de *software* não é correspondido pelo crescimento do investimento em segurança [6]. O número de vulnerabilidades em aplicações web tem vindo a aumentar devido ao desenvolvimento de novas técnicas de ataques informáticos. Em 2019 cerca de 12000 vulnerabilidades novas foram registadas na maior base de dados CVE (*Common Vulnerability Enumeration*), que é um dicionário público de cibersegurança. Em 2020, o número de vulnerabilidades reportadas atingiu um novo recorde [9, 10].

As aplicações web continuam a ser os alvos preferidos dos atacantes informáticos [2], e é reivindicado que cerca de 70% dos ataques são feitos à camada aplicacional e não à camada de rede [6]. Sabe-se também que o PHP é a linguagem mais utilizada para o desenvolvimento das aplicações web do lado do servidor [4].

Estas aplicações, cada vez mais, integram várias tecnologias e linguagens, fazendo com que o seu código seja mais complexo e consequentemente mais difícil de analisar estaticamente e de descobrir vulnerabilidades. Por seu turno, a representação do seu código numa linguagem intermédia, de baixo nível, remove esta complexidade, mas mantem os aspetos fundamentais e a semântica do código. Por estes motivos, é desejável a utilização de uma linguagem intermédia, de mais baixo nível, que permita detetar vulnerabilidades mais facilmente através de análise estática. O facto desta análise estática ser feita numa linguagem de mais baixo nível possibilita que futuramente as mesmas verificações de segurança possam ser feitas para outros programas escritos noutras linguagens.

A HipHop *assembly* (HHAS) é uma linguagem intermédia que permite representar PHP e Hack (linguagem usada no desenvolvimento do Facebook) a mais baixo nível, mantendo as características da linguagem de alto nível. A HHAS é a representação da linguagem HipHop *bytecode* (HHBC) consumida pela HipHop *virtual machine* (HHVM), que é legível e escrita por humanos. Ao contrário das declarações nativas do PHP, que apenas descrevem que ações devem ser realizadas, a HHAS não utiliza duas instruções diferentes que produzam o mesmo resultado, dizendo à máquina como essas ações devem ser realizadas. As operações de declaração, carre-

gamento, e alteração de variáveis são especificadas em [11] para linguagem HHBC e consequentemente para a HHAS, o que facilita a análise estática de um programa. Além disso, a HHAS introduz o conceito de "pilha de operandos", um espaço na pilha onde os dados são inseridos ou removidos consoante as diferentes operações vão sendo processadas. A representação em *bytecode* permite ainda minimizar o trabalho realizado em tempo de execução e que sejam feitas otimizações e análises antecipadamente [4, 12].

## 1.2 Objetivos

O objetivo desta dissertação é utilizar a linguagem de baixo nível HipHop *assembly* (HHAS) para detetar vulnerabilidades de injeção de SQL e XSS em aplicações web escritas em PHP através de análise estática de código e sistema de tipos. Esta análise tem em consideração as funcionalidades dinâmicas do PHP, tal como os rápidos ciclos de desenvolvimento que esta linguagem proporciona. Para atingir este objectivo, outros dois são colocados:

- O primeiro objetivo será analisar de que forma são exploradas estas duas classes de vulnerabilidades em aplicações web, estudar vários tipos de análises estáticas e ferramentas de deteção de vulnerabilidades de aplicações web. Para além disto, será importante identificar as características das linguagens dinâmicas, para poder adaptar estas técnicas e ferramentas ao nosso objectivo principal.

- O segundo objetivo será construir uma ferramenta que contenha um analisador estático de código, que permita apurar se o código de um programa é seguro em relação a um conjunto de vulnerabilidades de segurança predefinido. O analisador atuará sobre a linguagem intermédia. Ao ser realizada a análise a este nível, conseguimos que as linguagens de alto nível que sejam compiladas para esta linguagem de baixo nível também possam ser processadas pela ferramenta na descoberta de vulnerabilidades.

Por fim, a ferramenta será avaliada com código PHP para validar as suas funcionalidades.

## 1.3 Contribuições

As principais contribuições desta dissertação são as seguintes:

1. O estudo de diferentes análises de comprometimento e análises estáticas de código para a descoberta de vulnerabilidades;

2. A definição de uma análise de comprometimento, baseada numa análise de tipos, a partir de um sistema de tipos que definimos, a qual considera todas as características de uma linguagem imperativa e as seguintes funcionalidades da linguagem HHAS: *arrays* associativos, variáveis globais, *aliases* e variáveis acedidas dinamicamente;

3. A ferramenta DVHipHop que implementa a análise de comprometimento, sendo assim capaz de analisar aplicações web traduzidas para HHAS e identificar vulnerabilidades de injeção de SQL e de *Cross-Site Scripting*;

4. A avaliação da ferramenta através de testes. Nestes testes estão incluídos testes que foram desenvolvidos especificamente para esta ferramenta como testes de um *dataset* conhecido.

## 1.4 Estrutura do documento

Este documento encontra-se dividido da seguinte forma:

O Capítulo 2 apresenta duas das principais vulnerabilidades em aplicações web, dois tipos de linguagem de baixo nível utilizadas para traduzir código PHP. Também, introduz várias estruturas de dados e técnicas utilizadas para a análise estática e a deteção de vulnerabilidades estaticamente. Explica, ainda, algumas ferramentas que detetam estas vulnerabilidades. O Capítulo 3 apresenta de uma forma formal o Sistema de Tipos que foi definido e é utilizado para efectuar a análise de tipos que vai permitir encontrar programas vulneráveis. O Capítulo 4 apresenta a arquitetura da ferramenta desenvolvida e todos os seus componentes em detalhe. O Capítulo 5 descreve o sistema de avaliação criado para testar a ferramenta desenvolvida, bem como a avaliação da mesma. No Capítulo 6 é feita uma conclusão e são apresentados possíveis trabalhos futuros.

# Capítulo 2

# Trabalho relacionado

Neste capítulo são apresentados alguns temas que são importantes e servem de base para o desenvolvimento desta dissertação. São, também, apresentados alguns trabalhos relacionados que ajudam a perceber que decisões foram tomadas em contextos semelhantes, e as várias abordagens possíveis para resolver o mesmo problema. A Secção 2.1 descreve os ataques de injeção de SQL (SQLI) e de *Cross-Site Scripting* (XSS) que afetam aplicações *web*, e ilustra ataques que as exploram. A Secção 2.2 apresenta as técnicas de compilação de linguagens de alto nível e as características destas linguagens. A Secção 2.3 apresenta as várias técnicas utilizadas em análise estática para a deteção de vulnerabilidades em aplicações *web* e as várias ferramentas que as utilizam, bem como outro tipo de técnicas para encontrar essas vulnerabilidades.

## 2.1 Ataques

Esta secção foca-se em dois tipos de ataques a aplicações web: injeção SQL e *Cross-Side Scripting* (XSS). Estes dois tipos de ataques são os mais explorados em aplicações web [13] e podem ambos ser evitados através do reforço da validação das entradas dos utilizadores.

Na literatura existem abordagens para identificar e prevenir a exploração de ataques de SQLI e XSS, nomeadamente, através da utilização de código defensivo, análise estática, monitorização dinâmica e geração de testes. Apesar destas abordagens contribuírem para a identificação de vulnerabilidades, estas também oferecem uma oportunidade de melhoria. A abordagem de desenvolver código defensivamente está sujeita a erros e requer que o *software* já existente seja reescrito com bibliotecas seguras. As ferramentas de análise estática estão mais sujeitas a produzir falsos positivos. As ferramentas de monitorização dinâmica aumentam o tempo de execução e não detetam as vulnerabilidades até o programa ser executado. Os testes do tipo caixa branca (*white-box*) são desenvolvidos tendo em consideração o código

fonte, ao contrário dos testes do tipo caixa preta (*black-box*) que não têm qualquer conhecimento do código, mas que permitem a descoberta de novas vulnerabilidades [9], apesar de não as identificarem no código fonte.

O detetor de vulnerabilidades Ardilla é baseado em *dynamic taint analysis*[14]. Esta ferramenta marca dados vindos do utilizador como potencialmente *tainted*, segue esses dados tainted, e verifica se esses dados chegam a um *sensitive sink*. Uma *sensitive sink* é um ponto vulnerável no programa, ou seja, é uma função que ao receber dados maliciosos como argumento, pode gerar resultados e comportamentos inesperados. Por exemplo, a função *mysqli_query* de PHP, utiliza uma *String* como argumento para representar a *query* que irá enviar para a MySQL para ser executada na base de dados. Se uma *String* contruída a partir de dados *tainted* é passada para esta função, então o utilizador malicioso pode potencialmente fazer um ataque SQLI. O mesmo acontece se passarmos dados *tainted* a uma função que produza HTML para escrever resultados processados do lado do servidor, levando a ataques de XSS.

### 2.1.1 SQLI

Um pedido SQL permite desempenhar uma acção na base de dados. Estes pedidos muitas vezes utilizam argumentos que podem ser dados pelo utilizador através de, um parâmetro, um formulário, um parâmetro do URL ou uma *cookie* [15].

Uma injeção SQL ocorre quando dados introduzidos pelo utilizador são usados diretamente numa consulta SQL sem previamente serem sanitizados ou validados. Esses dados contêm meta carateres, como ' ou ", que alteram a *query* e que de uma certa forma enganam o interpretador SQL. Os principais motivos que levam ao SQLI reduzem-se às más práticas de programação, nomeadamente a falta de sanitização dos dados provenientes dos utilizadores, e ao modo de armazenamento utilizados pelas bases de dados [16, 17].

Como o atacante que estamos a considerar não conhece nada sobre a implementação da aplicação que pretende atacar, o ataque SQLI é feito da seguinte forma: Inicialmente o atacante estuda de que forma o programa usa determinados dados de entrada de utilizador num pedido SQL; O atacante faz isto submetendo vários valores e pedaços de código escolhidos cuidadosamente para o efeito; isto permite ao atacante explorar o pedido SQL submetendo um valor malicioso que tenha em conta o estudo feito para esse pedido [15].

Existem vários tipos de ataques de injeção SQL, sendo os tautológicos os mais usuais. Este tipo de ataque injeta código de forma a que o *statement* condicional seja sempre avaliado como verdadeiro, o que permite ignorar por exemplo, a autenticação do utilizador e extrair indevidamente dados da base de dados. Para isto o atacante tenta explorar os argumentos que da condição WHERE, transformando essa condição numa tautologia. Caso o WHERE esteja incluso na cláusula SELECT,

o ataque causará o retorno de todas as linhas da tabela desse pedido. O código PHP na Listagem 2.1, é um exemplo simples deste tipo de ataques. A variável `$u` recebe o nome do utilizador fornecido pelo utilizador (linha 1) e é inserido numa *query* (linhas 2-3) que permite obter a *password* do utilizador. O atacante pode inserir um nome de utilizador malicioso, tal como ' OR $1 = 1 - -$ , modificando a estrutura da *query* e obtendo todas as passwords dos utilizadores.

```
1 <?php
2   $u = $_POST['nomeutilizador'];
3   $q = "SELECT pass FROM utilizadores WHERE utilizador='".$u."'";
4   $query = mysql_query($q);
```

Listagem 2.1: Programa vulnerável a ataque de SQLI através de uma tautologia [2]

Isto é possível porque no código é feita uma simples concatenação de *Strings* para criar o pedido SQL. Portanto, qualquer valor escolhido é posto no final do *select statement*.

### 2.1.2 XSS

Um ataque de XSS resulta da utilização mal-intencionada de uma entrada de utilizador, a qual é usada pela aplicação para processar a página HTML seguinte. O atacante recorre engenharia social para convencer a vítima a seguir um URL disfarçado que contém código HTML/JavaScript malicioso. Caso a vítima clique no URL o navegador do utilizador mostra o HTML e executa o JavaScript que está contido no URL malicioso. O resultado deste ataque pode resultar no roubo de *cookies* do navegador ou de outras informações sensíveis do utilizador. Para prevenir este tipo de ataques os utilizadores devem verificar as ligações antes de as seguirem, e as aplicações devem rejeitar ou modificar os valores de entrada que contenham código de script. O código da Listagem 2.2 exemplifica um programa vulneravel a um ataque de XSS. O valor de entrada do utilizador se não for vazio, é guardado pela variável `$id_utilizador` (linhas 1-2), a qual, sem qualquer tipo de sanitização, é diretamente utilizada para contruir parte do código de um formulário em HTML através da função *echo* (linhas 3-4).

```
1     $id_utilizador = (isset ($_POST ['id_utilizador'])) ?
2         $_POST [ 'id_utilizador'] : '';
3     echo '<input type="hidden" nome="id_utilizador"
4             value ="'. $id_utilizador. '">';
```

Listagem 2.2: Programa vulnerável a ataque de XSS

## 2.2 *ByteCode*

Os analisadores estáticos podem realizar a análise de programas em linguagens de baixo nível para abstrair alguma complexidade das linguagens de alto nível. Nesta

secção apresentamos um compilador de código escrito em PHP e duas linguagens de baixo nível que representam programas de PHP em *bytecode*.

### 2.2.1 HHVM

A máquina virtual HipHop (HHVM) é um compilador JIT (just-in-time) em tempo de execução para PHP e Hack. O Hack foi criado pelo Facebook para ser um dialeto do PHP que o estende com mais algumas funcionalidades [18].

Um compilador JIT, faz a compilação de código em tempo de execução. Isto pode servir ambos os objetivos de um programador que são: a rápida iteração e a necessidade de grande desempenho em ambiente de produção, evitando ainda os riscos de correção de desenvolvimento e implantação em diferentes ambientes [12].

A HHVM é composta por dois componentes: o componente adiantado (*ahead-of-time*) e o componente em tempo de execução, que têm como interface entre si a linguagem de baixo nível HipHop *bytecode* (HHBC) [18]. A HHBC permite que um programa seja executado, sem que o componente de execução tenha acesso ao código fonte.

Inicialmente a HHVM utiliza o componente adiantado para analizar, otimizar e traduzir o código fonte para a linguagem de mais baixo nível HipHop *bytecode* (HHBC). Para isso este componente começa por fazer o *parse* do código fonte e produz uma AST, onde são efetuadas algumas otimizações, como propagação de constantes, *folding*, e *trait flattening*. De seguida o emissor de *bytecode* traduz a AST para HHBC, onde são feitas mais algumas otimizações pelo *HipHop Bytecode-to-Bytecode Compiler (hhbbc)* [18].

O componente de execução da HHVM utiliza uma pilha de avaliação onde são colocados e removidos valores temporários consoante as instruções HipHop *bytecode* (HHBC) consideradas [12]. Este componente tem dois mecanismos de execução: o interpretador de *bytecode* e o compilador JIT. Estes dois mecanismos são capazes de cooperar e mudar entre si virtualmente a qualquer instrução de *bytecode* [18].

O compilador JIT compila *tracelets* para código máquina representando as instruções HHBC e a informação de tipos da *tracelet* para uma representação intermédia de mais baixo nível, chamada HHIR. A HHIR é baseada em *Static Single Assignment* (SSA). É neste nível intermédio que são feitas as otimizações de compilador clássicas, é também feito relaxamento de guarda que é uma otimização específica das *tracelets* e ainda uma otimização que consiste na eliminação do contador de referências [18].

Uma das principais vantagens deste compilador da HHVM é ter acesso aos tipos dos dados das variáveis do programa e à abstração de uma *tracelet* (explicado na Secção 2.3.6), que lhe permite verificar os tipos de dados, o que não é possível verificar estaticamente, devido aos tipos dinâmicos do PHP e do Hack.

### HHBC

A lingagem HipHop *bytecode* (HHBC) [11] permite desassociar a fronteira entre o *front-end* que recebe o código fonte em PHP ou Hack e o *back-end* da HHVM que executa os programas [12]. A HHBC tem o formato apropriado para os programas serem consumidos pelos interpretadores e compiladores *just-in-time* pelo facto de utilizarem construtores mais simples para codificar excreções e declarações e por representar de forma mais intuitiva a ordem de execução do programa.

A HHBC foi desenhada para concretizar três objetivos: ser eficiente em tempo de execução, ser compatível com o PHP 5.5 possibilitando a compilação do código fonte escrito em PHP 5.5 para HHBC, e ser uma linguagem simples. Para isso, o design da HHBC deveria evitar funcionalidades que pudessem ser simplificadas ou removidas sem comprometer a compatibilidade, a eficiência de tempo de execução, e o design limpo do PHP 5.5.

Cada ficheiro de código fonte de um programa é compilado para uma unidade (unit) separada. Uma unidade é composta por *bytecode* que é guardado por um por *array* de *bytes* que representam uma sequência de instruções HHBC. Cada instrução pode ser codificada usando um ou mais bytes. O *bytecode* de uma unidade é dividido em seções, uma para cada função, que são delimitadas pelos meta-dados de cada unidade.

A HHBC representa o fluxo de execução usando uma pilha de *frames* que é denominada por "pilha de chamada". Uma *frame* é uma estrutura que logicamente contém um cabeçalho, um contador do programa (CP), um armazenamento de variáveis locais, um armazenamento de iteradores de variáveis, uma pilha de avaliação e uma pilha com informação dos parâmetros de cada função.

O *frame* no topo da pilha de chamada é referido como "*frame* corrente". O *frame* corrente representa a função que esta atualmente a ser executada. O contador do programa da *frame* corrente diz-se que é o "CP corrente". A qualquer instante, o CP corrente contem o endereço em *bytecode* da instrução em execução. Este contador quando atualizado aponta para a instrução seguinte, ou para outra instrução indicada pela instrução atual no caso de um salto.

A HHBC disponibiliza instruções especiais que permitem chamar e retornar uma função. Quando uma função é chamada, um novo *frame* é puxado para a pilha de chamada e é inicializado um CP a apontar para o ponto de entrada. O novo *frame* passa a ser o corrente, tal como o CP. Quando uma função retorna, o *frame* corrente é retirado do topo da pilha de chamada, o *frame* corrente passa a ser o anterior, tal como o CP. *Dispatcher* é o componente do mecanismo de execução responsável por tratar das chamadas e retornos das funções.

Cada instrução da HHBC é descrita pelo nome da instrução, seguido imediatamente por zero ou mais operandos, seguido por uma descrição da transição da pilha

na forma de $[xn, \cdots, x2, x1] \rightarrow [ym, \cdots, y2, y1]$, onde $[xn, \cdots, x2, x1]$ uma lista de valores, que descrevem o que a instrução consome da pilha de operandos e $[ym, \cdots, y2, y1]$ é a lista que descreve os valores que a instrução adiciona á pilha. Os elementos $x1$ e $y1$ representam o topo da pilha antes e depois da execução da instrução, respetivamente. Cada valor na pilha tem de ser uma célula ou uma referência que são representados na pilha por C e V, respetivamente. Cada elemento na transição da pilha pode ainda ter uma anotação do seu tipo como ilustra o exemplo da Listagem 2.3.

```
1 Add
2 [C:<T2> C:<T1>] -> [C:Dbl] se ( T1 == Dbl || T2 == Dbl)
3 [C:<T2> C:<T1>] -> [C:Int] se ( T1 != Dbl && T2 != Dbl)
```

Listagem 2.3: Especificação da transição do estado da pilha pela operação HHBC Add

Na Listagem 2.4 é apresentada uma tradução do programa da Listagem 2.1 para HipHop *assembly* (HHAS). A HHAS é uma representação da HHBC, também em *bytecode*, mas que é mais facilmente interpretada e escrita por um humano.

```
1  .main <"HH\\int" "HH\\int"  > main() {
2    String "_POST"
3    BaseGC 0 Warn
4    QueryM 1 CGet ET:"nomeutilizador"
5    SetL $u
6    PopC
7    String "SELECT pass FROM utilizadores WHERE utilizador='"
8    CGetL $u
9    Concat
10   String "'"
11   Concat
12   SetL $q
13   PopC
14   FPushFuncD 1 "mysql_query"
15   CGetL $q
16   FCall <> 1 1 - "" ""
17   SetL $query
18   PopC
19   Int 1
20   RetC
21 }
```

Listagem 2.4: Programa da Listagem 2.1 traduzido para HHAS

### 2.2.2 Zend Bytecode

De forma a suportar as características dinâmicas do PHP, a implementação padrão do PHP é feita pelo interpretador Zend. A escolha da utilização de um interpretador não é particular para o PHP, sendo que também é a abordagem mais comum usada para implementar outras linguagens dinâmicas como o Perl, Python, e Ruby [4].

O Zend é um interpretador de *bytecode*, o que significa que usa uma representação de programas de baixo nível (ou de linguagem intermédia), denominada de Zend *bytecode*. Apesar de não ser possível aceder diretamente a esse *bytecode* existem várias extensões externas que exportam esse *bytecode* para um formato que pode ser acedido [5].

Esta abordagem tem algumas semelhanças com um interpretador de uma AST, por ser um interpretador em andamento (walker interpreters). A primeira vez que um ficheiro é invocado, o Zend usa um analisador sintático para fazer a sua tradução para *bytecode*, para seguidamente interpretar estas instruções [4]. Este *bytecode* consiste em estruturas de dados chamadas *oparrays*, as quais são *arrays* que contêm a todas as operações de uma função juntamente com alguma informação auxiliar [5]. Por exemplo, a intrução `$a + $b` é representada por um *oparray*, contendo o opcode ADD e os dois operandos (`$a` e `$b`). Na sua configuração de maior desempenho, o Zend usa uma cache de *bytecode*, por exemplo, a *Advanced* PHP *Cache* (APC), que evita que sejam feitas análises e traduções repetidas entre execuções. Tal como outras linguagens dinâmicas, o Zend descobre e carrega vários componentes do programa (ex., funções, classes), consoante os ficheiros fontes são incluídos. Este procedimento é denominado de carregamento dinâmico. Os componentes do programa, incluindo classes, funções, variáveis, constantes, são mantidos em várias *lookup tables*. O carregamento dinâmico é particularmente custoso para classes, o que requer o carregamento dos seus métodos, das suas propriedades e constantes [4].

Durante a execução do *bytecode*, cada vez que o interpretador precisa de aceder a um símbolo, este usa o nome do símbolo para consultar as *lookup tables*, resultando num custo de tempo de execução do programa. Além do custo dos carregamentos dinâmicos e *lookups*, outra grande sobrecarga é o *dynamic typing*. Pelo facto das variáveis poderem ter diferentes tipos de dados em tempo de execução, estas são mantidas com valor genérico, chamadas de *z-values*. As instruções do Zend *bytecode* são maioritariamente sem tipo, mas ao serem interpretadas é realizada uma verificação de tipos dos operandos, de modo a serem executadas apropriadamente. Por exemplo, na instrução ADD é feita uma verificação para apurar se os operandos são números; caso não sejam, tenta convertê-los consoante o conjunto de regras de tipos [4].

## 2.3 Análise Estática

A análise estática é realizada por uma análise ao código sem que seja preciso executar a aplicação. Por este motivo esta análise considera todos os caminhos possíveis do programa e não apenas os que são percorridos em tempo de execução. Contudo, para desenvolver um analisador estático para uma linguagem dinâmica é preciso ter em conta que este tipo de linguagem toma algumas decisões em tempo de execução. Estas linguagens têm um fraco desempenho em relação a outras, sendo que a sua análise e correção devem ter sempre esse aspeto em consideração, bem como as possíveis alterações que sofrem as quais poderão exigir um grande esforço de manutenção.

Uma grande preocupação no desenvolvimento deste tipo de analisadores, é garantir que a análise efectuada por eles resulta no mínimo possível de falsos positivos (alarme sobre a existência de vulnerabilidades inexistentes) e de falsos negativos (vulnerabilidades não detectadas), isto porque os falsos positivos obrigam os programadores a verificarem vulnerabilidades que são inexistentes, enquanto os falsos negativos deixam o programa vulnerável. Este é um grande desafio visto que estes analisadores são forçados a fazer aproximações, mas não têm o conhecimento de toda a informação do código necessário para tal.

A análise estática pode ser realizada utilizando técnicas de *string pattern matching*, denominada de análise lexical. Esta serve apenas para fazer análises simples ao código, tal como procurar no código palavras específicas, ou de análise semântica, que envolve o processamento e adição de *tokens* aos ficheiros de código fonte para interpretar a semântica das instruções do programa. Com o uso destas análises é possível determinar se o código contém vulnerabilidades.

Para melhorar a precisão esta análise deve aproveitar as tecnologias de compilação, utilizando uma árvore de sintaxe abstrata (AST). É também importante não só ter em consideração a ordem como as operações são feitas através da AST, mas também o contexto em que essas operações são feitas.

### 2.3.1 Árvore de sintaxe abstrata

Um ponto de partida para fazer análise estática é utilizar uma estrutura de dados chamada árvore de sintaxe abstrata (AST). Numa AST, cada instrução é representada por um ramo da árvore. Cada ramo contém nós construtores para cada operação que a instrução contém (ex., adição, chamada de função) e abaixo deste, os nós filhos representam os parâmetros da operação. De uma forma geral, qualquer instrução pode ser manipulada através da criação de um operador para a sua re-

presentação e tratar como operandos os componentes semanticamente significativos dessa instrução.

A Figura 2.1 apresneta a AST da expressão 9 – 5 + 2 sendo a raiz representada pelo operador +. As subárvores da raiz representam as sub-expressões 9 - 5 e 2. O facto de 9 – 5 estar agrupado como um operando reflete o facto de a avaliação ser feita da esquerda para a direita em operandos com o mesmo nível de precedência. Como – e + têm a mesma precedência 9 – 5 + 2 é equivalente a (9 – 5) + 2 [19].

A utilização de uma AST é essencial quando é necessário saber por que ordem as instruções vão ser executadas. Esta estrutura permite fazer análises como *constant analysis* [4], *latent semantic analysis* [20], e construir um grafo de fluxo de controle (CFG) [20].

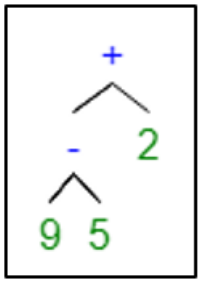


Figura 2.1: AST da expressão $9 - 5 + 2$

### 2.3.2 Grafo de fluxo de controle

O grafo de fluxo de controle (CFG) é um grafo que representa todos os caminhos pelos quais um programa pode ser percorrido durante a sua execução. Num CFG cada nó representa um bloco básico (ver Secção 2.3.6), as arestas direcionadas representam possíveis transferência de fluxo de controle de um boco básico para outro. Os CFG são normalmente usados para análise de fluxo de controle e dados, que é uma técnica para colecionar informação sobre o possível conjunto de valores calculados em vários pontos num programa ligados ao fluxo de execução do programa, tal como *if* e ciclos.

Normalmente durante a análise estática, é construído um CFG intra procedimental para cada função, antes de todos os CFG intra-procedimentais serem combinados num grafo procedimental (ICFG) que representa o fluxo de controle de todo o programa.

### 2.3.3 Grafo de chamada de funções

Um grafo de chamada de funções (CG) é um grafo que representa a dependência das funções num programa em que cada função é representada como um nó e as

invocações das funções são representadas por arestas.

### 2.3.4 Análise de fluxo de dados

Nesta secção são apresentadas três técnicas (*Reaching Definitions*, *Constants as a Feature* e *Taint Analysis*) que aproveitam a análise de fluxo de dados para extrair informação a partir do programa [20].

A análise de fluxo de dados (DFA) tem sido usada na otimização de compiladores durante décadas. A DFA é também utilizada para a deteção de vulnerabilidades [1, 8]. De um modo geral, o propósito da DFA é computar estaticamente informação como valores de variáveis, relações de *aliases* para cada ponto de um programa. Para isso, a DFA opera num CFG, que pode ser depois representado por um grafo de fluxo de dados em que cada nó tem a informação dos dados recolhidos para esse ponto do programa.

Por exemplo, a análise de constantes computa, para cada ponto do programa, os valores que uma variável pode ter. Para ilustrar como a análise de fluxo de dados é realizada, suponhamos um programa simples que usa apenas uma variável (v) e duas constantes (os inteiros 3 e 4) que v pode tomar. A Figura 2.2 ilustra o CFG deste programa. Nesta figura, cada nó do CFG é associado à informação final obtida pela DFA após a análise terminar. Os nós "Vazio" representam instruções vazias. É assumido que a condição do "*if*" não pode ser resolvida estaticamente por depender do valor de uma variável. O símbolo T ("*top*") é utilizado para uma constante desconhecida, o que indica que o valor exato da constante não pode ser determinado. Este é o caso de um ponto de entrada do utilizador do programa. Após ser feita a análise de constantes, a cada nó do CFG está associada a informação do valor de v que corresponde ao valor antes do nó ser executado. É importante notar que o valor exato da variável v a seguir ao "if" é desconhecida, pelo facto de não ser conhecido qual o caminho escolhido em tempo de execução.

#### *Reaching Definition*

Dada a definição de uma variável é importante saber quais das suas utilizações vão ser afetadas por essa definição. O inverso, em que determinada utilização de uma variável é afetada por determinadas definições, é também importante determinar. Estas relações de fluxo de dados, ou relações de "def-use", como são chamadas, podem ser deduzidas estaticamente. Para isto é necessário recolher informação para cada ponto do programa sobre *reaching definition* que corresponde às definições que foram feitas e não foram redefinidas, quando a execução do programa chega a esse ponto através de um determinado caminho.

No código da Listagem 2.5 podemos facilmente deduzir que todas as definições das variáveis `$x` e `$y` conseguem alcançar a linha 4. As definições na linha 1 e 2

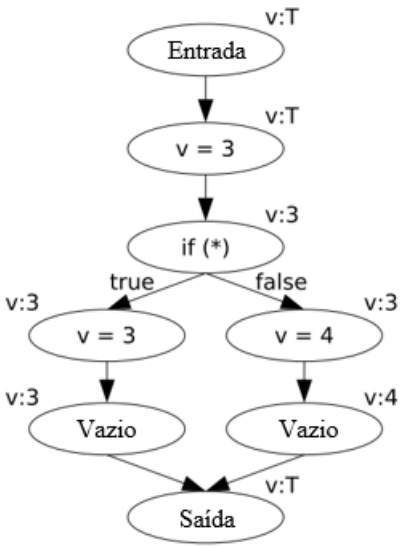


Figura 2.2: CFG de um programa simples [1]

chegam à linha 4 na primeira iteração. Na linha 5 apenas chegam as definições efetuadas nas linhas 1, 4 e 5, e não a da linha 2, porque a variável `$y` é redefinida na linha 4.

```
1       $x = 5;
2       $y = 1;
3       while ($x > 1){
4           $y = $x * $y;
5           $x--;
6       }
```

Listagem 2.5: Exemplo de programa para uso do def-use

O algoritmo de *reaching definition* mantém 4 conjuntos, que contêm nomes de variáveis, para cada linha do código. GEN[L] representa o conjunto de todas as definições dentro da linha L, que são visíveis imediatamente depois da linha L. KILL[L] é a união das definições feitas em todos os outros blocos básicos no grafo de fluxo, que são redefinidas na linha L. IN [L] representa todas as definições que são recebidas de linhas precedentes. OUT[L] representa a união de todas as definições criadas nessa linha e que não foram mortas nessa linha ou seja: OUT = GEN[L] $\cup$ (IN[L] $-$ KILL [L]). O algoritmo corre até que nenhum dos conjuntos OUT seja alterado numa iteração.

Após serem construídos os conjuntos de *Reaching Definition*, podem ser determinadas as correntes de *Use-Definition* para cada definição. As correntes UD consistem no uso de uma variável, e todas as definições dessa variável D, que conseguem alcançar essa utilização sem serem redefinidas [20].

### *Taint Analysis*

A análise de comprometimento (*taint analysis*) [1, 20] é uma das técnicas utilizadas para detetar vulnerabilidades do estilo *taint-style*. A *taint analysis* trata de identificar pontos de entrada de um programa por onde dados contaminados podem entrar e segue-os para verificar que não chegam a um ponto sensível do programa, denominado de *sensitive sink*, que pode dar origem a vulnerabilidades. Estes dados contaminados são criados estrategicamente com o intuito de comprometer o programa. Esta análise tem ainda de ter em conta que este estado contaminado (*tainted*) pode ser passado para outros dados através de uma atribuição (ex., `$v = $tainted`) ou removido caso os dados passem por um conjunto de operações que os tornam *untainted*, como operações aritméticas, utilização de *cast* ou a passagem por funções de sanitização (ex., a função *htmlentities* em PHP).

Algumas abordagens para fazer esta análise utilizam um conjunto predefinido de funções *sensitive sink* e *sanitization*, outras identificam-nas através de aprendizagem automática.

Para serem obtidos resultados precisos é necessário fazer outras análises como *alias analysis* (apresentado de seguida) e análise de constantes (explicado acima). A *alias analysis* permite propagar o estado *tainted* quando é efectuada uma atribuição a uma variável que tenha relações de *aliases* e a análise de contantes é fundamental para a análise de fluxo de dados.

### *Alias Analysis*

Para a obtenção de resultados precisos na deteção estática de vulnerabilidades, é necessário ter em conta as relações de *aliases* que as variáveis podem ter entre si. Sem este tipo de análise, a *taint analysis* pode gerar falsos positivos e falsos negativos. Duas variáveis são *aliases* num certo ponto do programa se os seus valores forem guardados no mesmo local da memória. Duas variáveis são *must-aliases* se forem *aliases* independentemente do caminho atual do programa em tempo de execução. Se estas variáveis forem *aliases* apenas em alguns caminhos do programa, e não para outros, são chamadas de *may-aliases* [1].

Em PHP *aliases* entre variáveis pode ser introduzido através do operador de referência "&". O código PHP da Listagem 2.6, mostra um exemplo simples da relação de *aliases* entre a variável `$a` e `$b` (na linha 2) e porque é que a *taint analysis* precisa de ter acesso a informação de *aliases*. Sem esta informação, a *taint analysis* não seria capaz de saber que a atribuição na linha 3 não afeta apenas `$a`, mas também a variável `$b`. Consequentemente, iriamos perder a informação de que a variável `$b` é *tainted*, levando à não detecção da vulnerabilidade de XSS na linha 4.

```
1      $b = 'untainted';
2      $a = & $b;
3      $a = $tainted
4      echo '<input type="hidden" nome="id_utilizador"
5              value ="'. $b. '">';
```

Listagem 2.6: Exemplo de *aliases* entre duas variáveis em PHP

Este tipo de análise [21] tem de ter em conta o âmbito das variáveis que têm uma relação de *aliases*. Por exemplo, o *aliases* de uma variável local com uma variável global só é válido dentro da função em que a variável local é definida, enquanto um *aliases* entre duas variáveis globais é sempre válido.

Pelo explicado acima, o principal problema surge com a análise entre procedimentos e a forma como são tratadas as chamadas a funções recursivas. Cada instância de chamada à função contém as suas próprias cópias das suas variáveis locais. Na maioria dos casos, não é possível decidir estaticamente quão profunda a corrente de chamadas recursivas pode ser, sendo que a profundidade pode depender de valores dinâmicos, como valores vindos de uma base de dados, ou de entradas de utilizador. Consequentemente, a análise estática enfrentaria uma quantidade infinita de variáveis que são inicializadas em cada recursão pelos parâmetros.

A solução para este problema é apresentada em [21], mostra que dentro de funções a análise só rastreia informações sobre variáveis globais e as próprias variáveis locais inicializadas pelos parâmetros.

### 2.3.5 *Type Checking*

Um tipo de dados numa linguagem de programação serve para atribuir às variáveis a informação de quais os tipos dos valores que estas podem tomar, e quais as operações que sobre estas podem ser realizadas. Um sistema de tipos evita "erros de tipo", ao examinar as instruções a efectuar sobre as variáveis de num programa. Por exemplo, um programa que atribui um booleano a uma variável do tipo inteiro é um erro em tempo de compilação. A verificação de tipos, tanto pode ocorrer em tempo de compilação (verificação de tipos estática) como em tempo de execução (verificação dinâmica de tipos), embora estas sejam completamente diferentes [6]. A linguagem PHP utiliza verificação dinâmica de tipos (a partir da versão 7) para fazer otimizações JIT (explicado na Secção 2.2.1).

No fragmento de programa PHP seguinte (Listagem 2.7), podemos ver que para saber o tipo da variável `$a`, o programa precisa de ser executado. Outro exemplo é a função *strlen* que ao ser passada uma *string* retorna um *integer* com o tamanho da *string*, mas quando é passado um *array* vazio retorna *null*.

```
1      $a = goldbachs_conjecture() ? 3.14159 : "uma string";
```

Listagem 2.7: Inferência de tipo indecidível

O facto de não serem utilizados tipos no PHP elimina a necessidade de compromisso antecipado com as estruturas de dados e suporta uma rápida evolução dos sistemas. Já para aplicações grandes torna-se complicado quando o código tem de ser revisto, especialmente para programadores que não estavam envolvidos na implementação original [22]. Apesar da análise estática de uma linguagem com tipos dinâmicos ser normalmente indecidível, é possível obter alguma informação de tipos através de heurísticas e de inferências, que são essenciais para a geração de código otimizado [4, 5, 18].

### 2.3.6 *Tracelets* e Blocos Básicos

#### Bloco Básico

Um bloco básico é um pedaço do programa onde não existem saltos ou não é alvo de saltos [19]. Uma vez que o código escrito numa linguagem como o *bytecode* é organizado em blocos básicos, podemos representar o fluxo de controle entre estes blocos através de um grafo de fluxo. Os nós do grafo de fluxo são blocos básicos e existe uma aresta do bloco B para o bloco C se e só se a primeira instrução do bloco C possivelmente sucede a última instrução do bloco B. Caso isso aconteça dizemos que B é o antecessor de C e C é o sucessor de B.

Geralmente, são adicionados dois nós, chamados de nó de entrada e nó de saída, que não correspondem a nenhuma instrução intermédia executável. Existe uma aresta entre o nó de entrada e o primeiro nó executável do grafo de fluxo, e existe uma aresta de qualquer nó que tenha uma instrução que possa ser a última a ser executada pelo programa, para o nó de saída.

#### *Tracelets*

Pelo facto da inferência de tipos ser a chave para otimizar o PHP foi criada a estrutura *tracelet* [12], a qual é similar a um bloco básico e especializada para um conjunto particular de tipos em tempo de execução, para os seus valores de entrada. A *tracelet* é uma região no programa fonte que é de entrada única e múltiplas saídas [12].

O compilador JIT da HHVM executa simbolicamente as instruções HHBC que compõem uma *tracelet*, passando por um caminho único, avançando até à análise de fluxo. Esta execução simbólica anota cada instrução HHBC com tipos de entrada e de saída. Caso haja algum tipo que não seja reconhecido na execução simbólica é resolvido observando o estado do programa em tempo de execução ou de compilação JIT. Após isto o JIT gera código e usa este estado observado para a previsão dos tipos à saída da *tracelet*. O código máquina gerado para uma *tracelet* precisa de ter primeiro uma guarda para assegurar que as precondições que guiaram à execu-

ção simbólica da *tracelet* são asseguradas em tempo de execução. Se a guarda for verdadeira, o compilador JIT otimiza o corpo da *tracelet* com toda a informação de tipos recolhida durante a execução simbólica, tal como a execução clássica das otimizações de compilação (propagação de constantes, eliminação de código morto, etc.). Caso a guarda seja falsa pela primeira vez, é criada uma nova *tracelet* para os tipos observados em tempo de execução. É então criada uma corrente de *tracelets* que a HHVM utiliza consoante os tipos de entrada. Esta corrente normalmente tem um ou dois candidatos, mas nos casos raros em que os tipos dos valores de entrada têm um elevado grau de polimorfismo foi definido que a corrente de *tracelets* teria no máximo 12 *tracelets* e no caso de falhar a HHVM iria recorrer ao interpretador de *bytecode*.

### 2.3.7 Ferramentas de deteção de vulnerabilidades

#### Dekant

A Dekant [2] é uma ferramenta que através de aprendizagem automática deteta vulnerabilidades como se fosse uma ferramenta de análise estática.

A abordagem utilizada segue o mesmo princípio utilizado no processamento da linguagem natural (NLP), em que é criado um modelo que classifica sequencias de observações. Isto é possível por existirem características em comum entra as linguagens naturais e as linguagens de programação (ex.: a existência de palavras, frases, uma gramática, e regras sintáticas).

Esta ferramenta utiliza o *Hidden Markov Model* (HMM) para classificar uma sequência de instruções de um trecho de um programa como sendo, ou não, vulnerável. O HMM é uma rede dinâmica Bayesian em que cada estado é representado por um nó que terá uma determinada probabilidade de ir para outro nó.

Para treinar o HMM foi feita uma seleção de trechos de código que representem bons exemplos de vulnerabilidades conhecidas, e que foram anotadas como sendo vulneráveis, ou não vulneráveis caso seja feita sanitização. Cada trecho tem início numa entrada de utilizador e termina numa *senvitive sink*. Todos os trechos de código foram traduzidos para *Intermediate Slice Language* (ISL) que se foca em traduzir instruções que possam estar associadas a vulnerabilidades ou preveni-las. O uso de uma linguagem intermédia permite que a ferramenta consiga analisar programas escritos em diferentes linguagens. Nesta tradução é também criado um mapa de variáveis que serve para manter conhecimento de quais as variáveis que dependem de valores de entrada e que os possam propagar para outras variáveis. Após a tradução para ISL cada trecho de código foi transformado em estados anotados manualmente para cada estado de cada observação (sequencia de instruções até essa instrução). Neste treino o modelo utilizou estas transições de estados para extrair conhecimento. Para descobrir uma vulnerabilidade num programa a ferramenta começa por extrair

todos os trechos do código que começam com uma entrada de utilizador e terminam numa *sensitive sink*. Para cada trecho faz a sua tradução para ISL e guarda todas as ocorrências das variáveis num mapa. Depois destes dois passos, para que o HMM consiga descobrir a melhor sequência de estados para representar a lista de instruções de um trecho, cada instrução de um trecho é classificada pelo HMM que utiliza a informação das variáveis das instruções anteriores (guardadas num mapa com a informação do seu estado malicioso), para emitir as probabilidades das classificações dessa instrução. A última observação feita à última instrução do trecho determina se esse trecho tem uma vulnerabilidade. Se uma vulnerabilidade for detetada, a sua localização é identificada no código fonte. Uma sequência de observações de instruções de um *slice-isl* pode começar em qualquer estado mas tem que terminar no estado *Taint* ou *N-Taint*.

### Pixy

A Pixy [1] é uma ferramenta *open source* que deteta vulnerabilidades estaticamente em código PHP, nomeadamente injeções SQL e XSS. Para isso é feita análise de fluxo de dados, em que são utilizadas duas análises em particular: *literal analysis* e *alias analysis* (ver Secção 2.3.4) que levam a resultados mais compreensivos e precisos, por fazerem com que a análise seja *interprocedural*, *flow-sensitive* e *context-sensitive*.

De modo a facilitar a análise estática e reduzir alguma complexidade da linguagem é feita a linearização de expressões arbitrárias profundas na linguagem original, tal como a redução de vários ciclos e ramos do programa como *foreach* e *switch* para combinações de *if* e *goto*. Esta representação intermédia, P-Tac, assemelha-se ao clássico código de triplo endereço (TAC). A TAC é uma linguagem idêntica ao *assembly*, é caraterizada por operações com no máximo três operandos e de forma geral representadas por "x = y op z". Por exemplo a operação "a = 1 + b + c" seria traduzida na correspondente sequência TAC: $t1 = 1 + b;\ t2 = t1 + c;\ a = t2$, as variáveis t1 e t2 são introduzidas temporariamente durante a tradução, e não aparecem em mais nenhum lugar no programa. Esta linguagem facilita a utilização de uma AST.

### WAP

A ferramenta WAP [8] não só deteta vulnerabilidades estaticamente em código PHP, como também prevê se os resultados da análise estática são falsos positivos ou não. Caso se confirme que de facto existe uma vulnerabilidade, esta ferramenta trata ainda da correção do código e da execução de testes para verificar se o problema foi resolvido. Além destes procedimentos automáticos, os programadores são informados das vulnerabilidades encontradas, as correções feitas a essas vulnerabilidades e os falsos positivos encontrados. Esta ferramenta está dividida em três módulos: o

*Code Analyser*, o *False Positives Predictor* e *Code Corrector*.

O *Code Analyser* analisa o código fonte para gerar uma AST, de seguida, um navegador passa sobre a AST para gerar uma *tainted symbol table* (TST) e *tainted execution path trees* (TEPT). Cada célula da TST é uma subárvore da AST, que representa operações importantes para serem analisadas. Para cada símbolo utilizado nessas operações é guardado o seu nome, número da linha, e o seu estado, que inicialmente é *tainted* apenas nas entradas de utilizador. Na TEPT cada ramo corresponde a uma variável *tainted* encontrada numa célula TST. Estes ramos contêm sub-ramos que indicam a linha onde essa variável foi definida como *tainted* tal como variáveis para as quais esse estado foi propagado a partir dessa definição. Existe ainda uma estrutura *untaited data* (UD), que guarda as variáveis que passaram de *tainted* a *untainted* através de *sanitization*, permitindo que o seu estado *tainted* não seja propagado.

Na *taint analysis* é feita uma travessia sobre a TST, em que são consultadas a TEPT e DT para encontrar informações do estado atual das variáveis utilizadas. Caso um símbolo a ser utilizado dependa de uma variável *tainted*, o seu estado vai ser propagado para a TST e para a TEPT. Por outro lado, se um símbolo com estado *tainted* na TEPT for utilizado como argumento de uma função de *sanitization*, esse símbolo passa a ser *untainted* e é adicionado à estrutura UD. Durante esta análise em qualquer altura que uma variável *tainted* passa por uma *sensitive sink*, é ativado o *false positives predictor*.

O *False positives predictor* utiliza prospeção de dados e aprendizagem automática para prever se as vulnerabilidades encontradas pelo *Code Analyzer* são falsos positivos.

No *Code Corrector* as vulnerabilidades que não foram descartadas, são então processadas utilizando a TST e a TEPT associada a essa vulnerabilidade, aplicando correções ao código de modo a remover a vulnerabilidade.

# Capítulo 3

# Sistema de Tipos

Neste capítulo é descrita a especificação de um analisador estático de comprometimento de segurança recorrendo a um Sistema de Tipos para um fragmento da linguagem HipHop *assembly*. Para isso, são apresentadas as instruções da linguagem HHAS que são utilizadas na análise de tipos. Esta linguagem é uma representação do HipHop *bytecode* [11], que é mais facilmente interpretada e escrita por um humano. As instruções consideradas são um subconjunto das instruções HHAS que capturam os principais problemas de segurança em aplicações web, tais como injeção SQL e XSS. Esta análise considera todas as características de uma linguagem imperativa, nomeadamente, a utilização de variáveis, os diferentes fluxos de controle e chamadas a funções. Para além disso são ainda consideradas as seguintes funcionalidades da HHAS: *arrays* associativos; variáveis globais; *aliases*; e variáveis acedidas dinamicamente.

As instruções da HHAS são especificadas formalmente na Secção 3.2.3. De seguida é apresentada a análise estática e o algoritmo de verificação de tipos na Secção 3.3. A semântica estática é apresentada de forma incremental, começando pelas instruções mais simples nas quais não é apresentado o contexto nem todas as funcionalidades consideradas pela análise. Nesta Secção algumas funcionalidades são apresentadas de modo informal, onde são mostrados alguns exemplos de código que ajudam a perceber como foi abordado o problema. A definição deste sistema de tipos foi feita com base em Freund e Mitchell [23].

## 3.1 HHAS

O compilador da HHVM (descrito na Secção 2.2.1) traduz um programa PHP ou Hack numa coleção de *Units* que representam cada um dos ficheiros do programa na linguagem HHAS. Aqui, é ilustrado esse processo de compilação através da tradução do programa PHP simples (na Listagem 3.1) para HHAS (na Listagem 3.2).

Na linha 7 da Listagem 3.1 a função `iniciaSessao` tem definido o *type hint* do

seu parâmetro. A declaração de um *type hint* requer que o interpretador verifique se um valor passado ou devolvido corresponde a esse tipo. Embora o tipo *sstring* não seja um tipo primitivo do PHP ou do Hack, na análise de tipos é sempre assumido que *sstring* é definido no programa como *alias* do tipo *string*. Isto permite que a análise considere o tipo *sstring* como uma *string* segura e o tipo *string* como uma *string* potencialmente maliciosa.

A HHVM é uma máquina de três pilhas: a pilha de avaliação, onde são adicionados e removidos valores consoante são executadas instruções; a pilha de *FID*, que contém identificadores de funções que estão prontas a serem chamadas; e a pilha de registos de ativação, que contém as sub-rotina ativas de um programa. A HHVM usa ainda um registo para aceder a posições de *arrays*.

```
1  <?php
2
3  $nomeUtilizador = htmlentities($_POST["nomeUtilizador"]);
4
5  iniciaSessao($nomeUtilizador);
6
7  function iniciaSessao(sstring $nomeUtilizador): void {
8    echo "Sessão iniciada com o utilizador " . $nomeUtilizador;
9  }
10 ?>
```

Listagem 3.1: Exemplo de programa PHP

O programa da Listagem 3.2 irá ser executado pela HHVM da seguinte forma:

As instruções da linha 2 à linha 9 fazem a chamada à função de sanitização `htmlentities`, uma função *built-in* da linguagem. A instrução `FPushFuncD` contém o número de parâmetros e o nome da função, que permitem identificar a função a ser chamada. Esta instrução adiciona o identificador da função `htmlentities` à pilha *FID*. A instrução na linha 3 adiciona a constante `_POST` ao topo da pilha de avaliação. Esta constante é usada pela instrução `BaseGC`, para cria um apontador para um *array*, sem alterar a pilha de avaliação. Este apontador é guardado num registo na máquina virtual. A instrução `QueryM` retira um valor da pilha de avaliação e executa a suboperação `CGet` sobre o *array* guardado no registo. A suboperação `CGet` é feita, por sua vez, sobre o elemento `nomeUtilizador` do *array*, que adiciona assim o valor guardado pela variável `$_POST["nomeUtilizador"]` ao topo da pilha de avaliação; e, por fim, a instrução `FCall` retira o identificador da função do topo da pilha de *FID* e faz a invocação da função que irá ser executada numa nova *frame*. Neste processo é retirado um valor da pilha de avaliação que é passado como parâmetro. A nova *frame* é adicionada ao topo da pilha de registos de activação, uma pilha de *frames*. Após esta função ser processada, é retirada a sua *frame* da pilha de registos de activação e é adicionado o valor devolvido ao topo da pilha de avaliação na *frame* da função que invocou. A instrução `SetL`, na linha 7, coloca o valor que está no topo da pilha na variável `$nomeUtilizador`, sem que esse valor seja retirado da pilha.

A operação `PopC` retira o valor no topo da pilha. As instruções da linha 9 à linha 11 fazem a chamada à função `iniciaSessao`, cuja definição se encontra da linha 17 à linha 25. Esta função, ao contrário da função `htmlentities`, recebe um parâmetro do tipo `sstring`, que está definido na linha 17. O processo de chamada desta função é igual ao anterior, mas a instrução `FCall` tem de verificar se o valor do parâmetro passado é uma `string`. Caso isso não aconteça é levantada uma exceção de *Type Mismatch*. Quando é criada e adicionada a nova *frame* ao topo da pilha de chamada é adicionado o parâmetro à tabela de símbolos dessa *frame*.

A função `iniciaSessao` é então executada pela HHVM da seguinte forma: a primeira instrução adiciona uma *string* ao topo da pilha; e a instrução seguinte adiciona o valor da variável `$nomeUtilizador` ao topo da pilha, que neste caso também é uma *string*. Estas duas *strings* são concatenadas pela instrução `Concat`, que retira essas duas *strings* da pilha de avaliação e adiciona a *string* concatenada à pilha. A instrução `print` é uma instrução que traduz especificamente a função *sensitive sink echo*. Esta função recebe um parâmetro e, portanto, a instrução *print* consome a *string* no topo da pilha, imprime-a para o ecrã e adiciona um inteiro com o valor 1 ao topo da pilha, caso a instrução seja executada com sucesso. O valor adicionado pela instrução `print` é removido da pilha pela instrução `PopC` seguinte. Como esta função é *void*, é devolvido o valor *Null*. Este é adicionado na linha 23 à pilha de avaliação e devolvido na instrução seguinte, `Retc`.

```
1  .main <"HH\\int" "HH\\int"  > main() {
2    FPushFuncD 1 "htmlentities"
3    String "_POST"
4    BaseGC 0 Warn
5    QueryM 1 CGet ET:"nomeUtilizador"
6    FCall <> 1 1 - "" ""
7    SetL $nomeUtilizador
8    PopC
9    FPushFuncD 1 "iniciaSessao"
10   CGetL $nomeUtilizador
11   FCall <> 1 1 - "" ""
12   PopC
13   Int 1
14   RetC
15 }
16
17 .function <"HH\\void" N  > iniciaSessao(<"sstring" "sstring"  >
       $nomeUtilizador) {
18   String "Sessão iniciada com o utilizador "
19   CGetL $nomeUtilizador
20   Concat
21   Print
22   PopC
23   Null
24   RetC
25 }
```

Listagem 3.2: Tradução do programa da Listagem 3.1 numa *Unit* em HHAS

### 3.1.1 Sintaxe da linguagem HipHop *assembly*

A Figura 3.1 representa a sintaxe da HHAS. De seguida são descritas, de forma resumida, as instruções desta sintaxe, que estão organizadas em 9 secções. Na Figura 3.1, $x$ representa o nome de uma variável, *int* representa um inteiro e uma *string* é representada por $s$ ou pelas palavras a negrito: ***LocalizaçãoFicheiro***; ***NomeFicheiro***; ***NomeF***; e ***NomeP***.

Programa ::= $\left\{ \text{IdArrayVazio?} \right\}$
$\left\{ \text{Função} * \right\}$

IdArrayVazio ::= .adata A_0 =$''''''$ $a : 0 : \{\}''''''$;

Função ::= **NomeF** `THint_Ret` : $\Big( (\texttt{PorRef?}\ \textbf{NomeP} : \texttt{THint}) * \Big)$
$\left\{ \texttt{Instrução} * \right\}$

`PorRef` ::= &

`Instrução` ::= (Instruções básicas) (1)
`Nop` | `PopC` | `PopV` | `Dup`

(Intruções de literais) (2)
| `Null` | `True` | `False` | `Int` ***int***
| `String` **s**

(Instruções de Operadores) (3)
| `Pow` | `Sqrt` | `Mul` | `Div` | `Mod` | `Add` | `Sub` | `Concat`
| `BitAnd` | `BitOr` | `BitXor`
| `Shl` | `Shr`
| `Xor` | `Same` | `NSame` | `Eq` | `Neq` | `Lt` | `Lte` | `Gt` | `Gte`
| `Cmp` | `Abs` | `Floor` | `Ceil` | `Not` | `BitNot`
| `CastBool` | `CastInt` | `CastString`
| `DblAsBits` | `Print`

(Instruções de Fluxo de Controle) (4)

| `Jmp` $l$ | `JmpZ` $l$
| `RetC`
| $l$

(Instruções Get) (5)
| `CGetL` **x** | `CGetL2` **x** | `PushL` **x**
| `VGetL` **x** | `VGetN` | `VGetG`
| `CGetN` | `CGetG`

(Instruções Isset e Type querying) (6)
| `IssetC` | `IssetL` **x** | `IssetG`
| `EmptyL` **x** | `EmptyG`
| `IsTypeC` | `IsTypeL` **x**

(Instruções de atribuição) (7)
| `SetL` **x** | `SetOpL` **x** $\boldsymbol{\omega}$ | `SetN`
| `SetG` | `SetOpG` $\boldsymbol{\omega}$
| `IncDecL` **x** $\boldsymbol{\delta}$ | `IncDecG` $\boldsymbol{\delta}$
| `BindL` **x** | `BindG`
| `UnsetL` **x** | `UnsetG`

(Instruções de chamada a uma função) (8)
| `FPushFuncD` **int nomeFunção**
| `FCall` <> **int 1** " " " "

(Instruções de *arrays* associativos) (9)
| `Array` @A_0
| `BaseGC` ***int*** | `BaseL` **x**
| `QueryM` ***int*** $\boldsymbol{\rho}$ $\boldsymbol{\kappa}$ **x** | `SetM` ***int*** $\boldsymbol{\kappa}$ **x**

$l \in$ `Etiqueta` ::= `L` ***int***

$\omega \in$ SubOpBin ::= Add | Sub | Mul | Concat | Div
| Pow | Mod | AndEqual | OrEqual | Xor | Shl | Shr

$\delta \in$ SubOpIncDec ::= PreIncO | PreDecO | PosIncO | PosDecO

$\rho \in$ QueryOp ::= `CGet` | `Isset` | `Empty`

$\kappa \in$ TChaveMembro ::= `ET` | `EI` | `EL`

Figura 3.1: Sintaxe da HHAS

Analisemos as características das instruções dos diferentes grupos da Figura 3.1.

As instruções básicas (grupo (1)) removem valores da pilha de avaliação.

As instruções literais (grupo (2)) são utilizadas para adicionar valores constantes na pilha.

As instruções de operadores (grupo (3)) permitem fazer operações com um ou dois valores na pilha de avaliação (ex., a soma entre dois valores é representada pela operação `Add`, que retira dois valores do topo da pilha).

As instruções de fluxo de controle (grupo (4)) utilizam saltos para etiquetas para definir, e permitir que sejam seguidos, os diferentes caminhos do programa.

As instruções *Get* (grupo (5)) permitem adicionar o valor de cada variável ao topo da pilha de avaliação, exceto a instrução `CGetL2` que adiciona o valor da variável à segunda posição da pilha. As instruções nas duas primeiras linhas vão buscar o valor da variável local definido por $x$; as instruções da terceira linha vão buscar o endereço de memória onde está guardado o valor correspondente à variável. As operações com N representam variáveis locais acedidas dinamicamente (variáveis de variáveis), enquanto as que têm um G representam o acesso a variáveis globais tanto de forma estática com dinâmica. As variáveis acedidas pelas operações com N e G são definidas pelo valor no topo da pilha.

As instruções `IssetC` e *Type querying* (grupo (6)) permitem saber se uma variável já foi definida, se tem algum valor e qual é o seu tipo.

As instruções de atribuição (grupo (7)) tratam de alterar o valor de uma variável: As operações `SetL` e `SetG`adicionam o valor no topo da pilha a uma variável local ou a uma variável global, respectivamente. As operações `SetOpL`, `SetOpG`, `IncDecL` e `IncDecG` alteraram o valor de uma variável e utilizam o seu valor juntamente com outro valor retirado da pilha para executar uma operação que é representada por $\boldsymbol{\omega}$ (uma das operações descritas no final da Figura 3.1). As operações `BindL`, `BindG`, `UnsetL` e `UnsetG` são operações que alteram a relação de *aliases* entre variáveis locais e variáveis globais.

As instruções de chamada a uma função (grupo (8)) funcionam da seguinte forma: inicialmente é utilizada a instrução `FPushFuncD` para preparar a chamada à função. Esta instrução contém o número de parâmetros e o nome da função que são adicionados ao topo da pilha de `FID`, no formato `FID_AS` como é representado na Figura 3.2. De seguida são utilizadas outras operações que definem que parâmetros irão ser passados em tempo de execução. Por fim, é utilizada a instrução `FCall`, que retira da pilha de `FID` o identificador da função a ser chamada, retira da pilha de avaliação o número de valores definidos por *int* (segundo argumento) e chama a função passando esses valores como parâmetros. O nosso sistema de tipos considera apenas o segundo argumento desta instrução.

As intruções sobre *arrays* associativos (grupo (9)) permitem inicializar um *array*, utilizá-lo e executar operações sobre as suas posições. Consequentemente, premitem utilizar as variáveis predefinidas, "super"globais de entradas de utilizador. Na análise de tipos só consideramos que um *array* é inicializado vazio e, portanto, consideramos apenas um identificador `A_0`, que identifica um *array* vazio representado no `Programa` por `IdArrayVazio`. As instruções `Base` servem para criar apontadores para *arrays*. Estes apontadores são guardados num registo e utilizados pelas instruções finais como, a `QueryM` e a `SetM`, que servem para fazer operações sobre essas variáveis. A instrução `BaseL` cria um apontador para a variável local $x$, enquanto a operação `BaseGC` cria um apontador para a variável definida pelo valor contido na pilha no índice *int*. As operações finais definem o acesso a uma posição do *array* através do símbolo $\boldsymbol{\kappa}$, que representa a forma de acesso, e $x$, que consoante $\boldsymbol{\kappa}$ pode ser um índice ($\boldsymbol{\kappa} = \texttt{EI}$), uma chave constante ($\boldsymbol{\kappa} = \texttt{ET}$), ou um nome de uma variável ($\boldsymbol{\kappa} = \texttt{EL}$). Ambas as operações finais, `QueryM` e `SetM`, retiram ***int*** valores da pilha de avaliação antes de adicionarem o resultado da operação ao topo da pilha. A operação final `QueryM` representa uma operação diferente consoante o valor do símbolo $\boldsymbol{\omega}$ que pode ser um dos três contidos em `QueryOp`. A operação `QueryM` permite executar a operação definida por $\boldsymbol{\omega}$ sobre a posição de um *array*. A operação final `SetM` permite atribuir o valor no topo da pilha a uma posição de um *array*.

A sintaxe de tempo de execução é representada na Figura 3.2. Esta sintaxe não é utilizada para escrever o programa, mas é necessária para descrever a semântica operacional. Nesta sintaxe o `VarID` premite identificar uma variável. Para isso, cada `VarID` tem uma `String` que é o nome da variável, um `TVariavel` para identificar se é uma variável `local` ou `SuperGlobal`. Na semântica operacional `FID_AS` é o identificador de uma função, que é composto pelo número de parâmetros $np$ e o nome da função $nf$. Na semântica estática uma função é identificada por *FID* que é composto pelo tipo da função $tf$ e pelo seu nome $nf$.

$$
\begin{array}{lrl}
\varphi \in & \texttt{VarID} ::= & (\texttt{nome} : \boldsymbol{s}\ \texttt{tvariavel} : \text{TVariavel}) \\
\\
\phi \in & \text{TVariavel} ::= & \texttt{local} \mid \text{SuperGlobal} \\
& \texttt{SuperGlobal} ::= & \epsilon \mid \texttt{globals} \mid \texttt{TvEU} \\
& \texttt{TvEU} ::= & \texttt{get} \mid \texttt{post} \mid \texttt{files} \mid \texttt{cookie} \mid \texttt{request} \\
& & \mid \texttt{session} \mid \texttt{env} \\
\\
\mathit{fid} \in & \texttt{FunçãoID} ::= & \texttt{FID\_AS} \mid \mathit{FID} \\
& \texttt{FID\_AS} ::= & \{|np : \boldsymbol{int},\ nf : \boldsymbol{s}|\} \\
& \mathit{FID} ::= & \{|tf : \texttt{TF},\ nf : \boldsymbol{s}|\} \\
\zeta \in & \texttt{TF} ::= & \texttt{main} \mid \texttt{função}
\end{array}
$$

Figura 3.2: Sintaxe de tempo de execução

### 3.1.2 Tipos para a HHAS

A gramática definida abaixo, na Figura 3.3, descreve os tipos utilizados na semântica operacional.

As instruções da HHAS podem adicionar, ou remover, valores da pilha de avaliação. Estes valores podem ser de duas variedades representadas por `FD` (*flavor descriptor*), onde `C` é uma célula (*cell*) e `V` é uma referência (*ref*). Uma célula é uma estrutura que contém um valor e o identificador do tipo. Uma referência é um apontador para uma célula.

Como a HHAS é uma linguagem dinamicamente tipificada, tal como o PHP, é feita uma análise de tipos em tempo de execução. Por este motivo, na semântica operacional cada valor apresentado numa transição da pilha pode ser acompanhado por um tipo `T_AS`.

A gramática seguinte, da Figura 3.4, é utilizada para gerar os tipos utilizados na análise de tipos. Estes tipos são identificados pelas letras apresentadas na figura, por $f$ e por $e$.

O tipo HipHop (`THH`) é o tipo contido pela pilha de tipos da análise estática (na Secção 3.3). Ao fazer a análise de tipos este tipo pode conter:

- Um `T`, que define o tipo do resultado de uma operação ou o tipo de um valor literal.
- Um `Untainted` e uma `String`, que representa um tipo mais geral `Untainted` de uma `String` constante em que é conhecido o seu valor constante $\boldsymbol{s}$, que irá ser utilizado na análise para aceder estaticamente a variáveis globais.

- Um `T` e um `VarID`, no caso de ser passada uma variável por referência, a variável é identificada por `VarID`e o seu tipo por `T`.
- Um `TCaixa` e um `VarID`, no caso de ser necessário criar uma relação de *aliases*, onde `VarID` é o identificador de uma variável e `TCaixa` é o tipo dessa variável que contem um tipo `T`.

O tipo `T` é o tipo que representa o estado malicioso de um valor primitivo ou de um *array*. Um tipo `T` pode ser:

- `TP`, que pode ser `Tainted` ou `Untainted`, que define se valores primitivos têm um tipo malicioso ou não.
- `Untainted`, que representa um *array* em que todas as posições são `Untainted`.
- `TaintedArray Array`, que representa um *array* com todas as posições não definidas como `Tainted`. Este tipo é considerado `Tainted` no caso de ser utilizado numa operação entre primitivos, como por exemplo numa concatenação.

Na análise estática assumimos sempre que o sistema de tipos da linguagem HHAS tem o tipo *sstring* que identifica uma *string* como segura (por oposição à potencialmente maliciosa).

$$
\begin{aligned}
\nu \in \quad & \texttt{FD} ::= C \mid C : \texttt{T\_AS} \mid V \\
& \texttt{T\_AS} ::= \texttt{Str} \mid \texttt{Null} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{Dbl} \mid \texttt{Arr} \\
\\
\eta \in \quad & \texttt{THint} ::= \epsilon \mid \texttt{string} \mid \texttt{sstring} \mid \texttt{integer} \mid \texttt{boolean} \\
& \qquad \mid \texttt{array} \mid \texttt{untainted} \mid \texttt{tainted} \\
\\
r \in \quad & \texttt{THint\_Ret} ::= \texttt{THint} \mid \texttt{void} \\
\\
\alpha \in \quad & \texttt{Lista-THint} ::= \epsilon \mid \texttt{THint} \cdot \texttt{Lista-THint} \\
\\
\varkappa \in \quad & \texttt{Lista-NomeParâmetros} ::= \epsilon \mid \boldsymbol{x} \cdot \texttt{Lista-NomeParâmetros} \\
\\
& \texttt{Função\_AS} ::= \langle \texttt{Lista-THint}, \texttt{Lista-NomeParâmetros}, \\
& \qquad \texttt{THint\_Ret} \rangle
\end{aligned}
$$

Figura 3.3: Sintaxe dos Tipos

$$
\begin{array}{rrl}
\mu \in & \texttt{THH} ::= & \texttt{T} \mid (\texttt{Untainted } \boldsymbol{s}) \mid \texttt{T VarID} \\
& & \mid \texttt{TCaixa VarID} \\
\psi \in & \texttt{TCaixa} ::= & \langle \texttt{T} \rangle \\
\tau \in & \texttt{T} ::= & \texttt{TP} \mid \texttt{TArray} \\
& \texttt{TP} ::= & \texttt{Untainted} \mid \texttt{Tainted} \\
& \texttt{TArray} ::= & \texttt{UntaintedArray} \mid \text{TaintedArray} \\
& \text{TaintedArray} ::= & \texttt{TaintedArray Array} \\
\varpi \in & \texttt{Array} ::= & \epsilon \mid \text{ArrayAssociativo<(s TP)*>} \\
\varsigma \in & \texttt{ListaT} ::= & \epsilon \mid \texttt{T} \cdot \texttt{ListaT} \\
\beta \in & \texttt{Lista-THH} ::= & \epsilon \mid \texttt{THH} \cdot \texttt{Lista-THH} \\
\gamma \in & \texttt{Retorno} ::= & \texttt{THint} \mid \texttt{T} \\
\kappa \in & \texttt{Lista-Booleanos} ::= & \epsilon \mid \texttt{Booleano} \cdot \texttt{Lista-Booleanos} \\
f \in & \textit{Função} ::= & \langle \textit{fid} : \textit{FID},\ \textit{ltparams} : \texttt{Lista-THint}, \\
& & \textit{lparams} : \texttt{Lista-NomeParâmetros}, \\
& & \textit{lparamsPRef} : \texttt{Lista-Booleanos}, \\
& & \textit{ret} : \texttt{Retorno},\ \textit{tipoExt} : \textit{Exterior} \rangle \\
e \in & \textit{Exterior} ::= & \langle \textit{varExt} : \text{Mapa<}\texttt{VarID}, \texttt{T}>,\ \textit{taExt} : \text{TabelaAliases}, \\
& & \textit{uExt} : \text{Lista<}(\texttt{VarID}, \texttt{T})>,\ \textit{adtExt} : \texttt{Booleano} \rangle
\end{array}
$$

Figura 3.4: Tipos

## 3.2 Semântica Dinâmica

Esta secção apresenta o modelo formal de execução de programas HHAS. A Secção 3.2.1 descreve a representação dos programas. A Secção 3.2.2 mostra o modelo utilizado como máquina de estados na semântica operacional e a Secção 3.2.3 descreve

a semântica das instruções de código fonte.

### 3.2.1 Ambiente

O ambiente para modelar a execução de um programa HHAS encontra- se na Figura 3.5. O ambiente $P$ é um mapa parcial de identificadores de funções para as suas instruções.

$$P : \texttt{FID\_AS} \rightharpoonup \langle \text{Instrução}^{+} \rangle$$

Figura 3.5: Formato do ambiente de um programa HHAS

Num ambiente a informação sobre uma função *fid* é acedida via $P[\mathit{fid}]$. Se $\Gamma[\mathit{fid}] = \langle I \rangle$ para determinada função, então $Dom(I)$ é o conjunto $\{1, \ldots, n\}$ de todos os endereços das instruções de um função, e $I[i]$ é a $i$-ésima instrução que pode ser acedida através da notação $\mathit{fid}_P[i]$.

### 3.2.2 Máquina de Estados

O estado de execução da HHAS é uma configuração $C = A;\ F;\ ts_0$, onde, $A$ é uma pilha de registos de ativação para cada *frame*, $F$ é um mapa de `FID_AS` para `Função_AS` e $ts_0$ é o mapa de identificadores de variáveis globais para variáveis. Um registo de ativação é definido da seguinte forma: $a ::= \langle \mathit{fid},\ pc,\ ts,\ p,\ rbm,\ pf \rangle$ Cada um dos elementos da configuração têm o seguinte significado:

- *fid*: identificador da função (`FID_AS`) correspondente a esse *frame*.
- *pc*: número da instrução que está a ser executada.
- *ts*: tabela de símbolos que consiste num mapa de `VarID` para V.
- $p ::= \epsilon \mid \texttt{FD} \cdot p$. Pilha de avaliação que contem valores `FD`.
- *rbm*: registo da base de membros que guarda o `VarID` do *array* que vai ser utilizado na operação final `QueryM` ou `SetM`.
- $pf ::= \epsilon \mid \texttt{FID\_AS} \cdot pf$. Pilha de funções que contem identificadores `FID_AS`.

### 3.2.3 Semântica Operacional

Tal como é feito em [19] a execução de um programa é especificada de uma forma padrão por uma semântica operacional (SO), em que o comportamento da análise perante uma instrução é representado por transições de uma máquina de estados. As regras semânticas estão representadas da Tabela 3.2.1 à Tabela 3.2.8. Cada linha dessas tabelas descreve as condições sobre as quais um programa representado por $P$ pode transitar de uma configuração $C_0$ para a configuração $C_1$. A primeira

coluna indica a forma da instrução capturada pela regra. Se a instrução prestes a ser executada satisfaz todas as condições nas colunas seguintes, então uma transição pode ser feita de uma configuração que corresponda a $C_0$ (penúltima coluna) para a configuração $C_1$ (última coluna). Na transição entre configurações os elementos que forem representados por *underscore* não são alterados com a transição. São ainda utilizadas funções na mesma coluna da condição para descrever alterações feitas à configuração $C_1$ que decidimos não formalizar, mas que descrevemos de modo informal.

Para especificar estas regras são usadas várias convenções e notações. Por exemplo, numa configuração inicial é usada a notação $c_1 : \texttt{Str} \cdot p$, que mostra que a célula $c_1$ está no topo da pilha que tem um valor do tipo `Str`. Se esta configuração passar para uma com a pilha $c_2 \cdot p$, significa que o resto da pilha $p$ permanece igual, mas a célula no topo da pilha deixou de ser $c_1$ e passou a ser a célula $c_2$ que tem um valor com um tipo indefinido.

Se a célula $c_1$, na primeira notação, não tivesse um valor do tipo `String` quando foi adicionado ao topo da pilha, a notação indica que é feito o *cast* para o tipo ***`Str`***.

Para representar a alteração do valor de uma variável são usadas duas notações diferentes:

$$\begin{cases} ts[x \mapsto v] = & \text{A seta "}\mapsto\text{" representa que a variável } x \text{ ficou com a referência } v \\ ts[x] = & \text{Representa a referência da variável } x \\ ts[x \twoheadrightarrow c] = & \text{A seta "}\twoheadrightarrow\text{" representa que a célula da referência da variável } x \\ & \text{passou a ter o valor da célula } c. \text{ No caso da variável } x \text{ ainda não ter} \\ & \text{sido inicializada, é criada uma referência para } x \text{ com uma célula} \\ & \text{com o mesmo valor da célula } c. \\ {}^{*}ts[x] = & \text{Representa o valor da célula da referência da variável } x \end{cases}$$

Estas duas notações são importantes para mostrar como são alterados os valores das variáveis mantendo as relações de *aliases*.

A notação $ts[pNomes_0, \ldots, pNomes_{|pNomes|} \to s1]$ é usada como abreviatura para $ts[pNomes_0, \to s1_0][pNomes_1, \to s1_1] \cdots [pNomes_{|pNomes|}, \to s1_{|s1|}]$ na Tabela 3.2.7, que formaliza a chamada a uma função. Nesta formalização não são tidos em conta os parâmetros passados por referência.

Tabela 3.2.1: Instruções Básicas da SO

$P \vdash C_0 \to C_1$

| $\mathit{fid}_P[pc]$ | **Cond** | $C_0$ | $C_1$ |
|---|---|---|---|
| `Nop` | | $\left\langle \mathit{fid},\ pc,\ \_,\ \_,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \mathit{fid},\ pc+1,\ \_,\ \_,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `PopC` | | $\left\langle \_,\ pc,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `PopV` | | $\left\langle \_,\ pc,\ \_,\ v \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `Dup` | | $\left\langle \_,\ pc,\ \_,\ c_1 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ c_1 \cdot c_1 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |

Tabela 3.2.2: Instruções de Literais da SO

$P \vdash C_0 \to C_1$

| $\mathit{fid}_P[pc]$ | **Cond** | $C_0$ | $C_1$ |
|---|---|---|---|
| `Null`<br>`True`<br>`Int` **int**<br>`String` **s** | | $\left\langle \_,\ pc,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |

Tabela 3.2.3: Instruções de Operadores da SO

$P \vdash C_0 \to C_1$

| $\mathit{fid}_P[pc]$ | Cond | $C_0$ | $C_1$ |
|---|---|---|---|
| Add | | $\left\langle \_,\ pc,\ \_,\ c_1 : \texttt{Arr} \cdot c_2 : \texttt{Arr} \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ (c_1 +_{\texttt{Arr}} c_2) \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| | $t1 = \texttt{Dbl} \vee$ $t2 = \texttt{Dbl}$ | $\left\langle \_,\ pc,\ \_,\ c_1 : t1 \cdot c_2 : t2 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ (c_1 +_{\texttt{Dbl}} c_2) \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| | $t1 \neq \texttt{Dbl} \wedge$ $t2 \neq \texttt{Dbl} \wedge$ $(t1 \neq \texttt{Arr} \vee$ $t2 \neq \texttt{Arr})$ | $\left\langle \_,\ pc,\ \_,\ c_1 : t1 \cdot c_2 : t2 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ (c_1 +_{\texttt{Int}} c_2) \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| Abs | $t = \texttt{Int} \vee$ $t = \texttt{Dbl} \vee$ $t = \texttt{Bool}$ | $\left\langle \_,\ pc,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ \lvert c \rvert : \texttt{Int} \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| Concat | | $\left\langle \_,\ pc,\ \_,\ c_1 \cdot c_2 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_, (c_1 \ {}_{\texttt{Str}}\cdot\ c_2) \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| CastInt | | $\left\langle \_,\ pc,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_, ((int)c) \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| Print | | $\left\langle \_,\ pc,\ \_,\ c_1 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ c_2 : \texttt{Int} \cdot\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |

Tabela 3.2.4: Instruções de Fluxo de Controle da SO

$P \vdash C_0 \to C_1$

| $\boldsymbol{fid_P[pc]}$ | **Cond** | $\boldsymbol{C_0}$ | $\boldsymbol{C_1}$ |
|---|---|---|---|
| `Jmp` $l$ | $linhaEtiqueta(l) = c$ | $\left\langle \_,\ pc,\ \_,\ \_,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ c,\ \_,\ \_,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `JmpZ` $l$ | $linhaEtiqueta(l) = c$ | $\left\langle \_,\ pc,\ \_,\ 0 : \texttt{Bool} \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ c,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| | $c \neq 0$ | $\left\langle \_,\ pc,\ \_,\ c : \texttt{Bool} \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| $l$ | | $\left\langle \_,\ pc,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `RetC` | | $\left\langle \_,\ \_,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ \_$ | $\epsilon;\ \_;\ \_$ |
| | $A \neq \epsilon$ | $\left\langle fid,\ \_,\ \_,\ c \cdot p_1,\ \_,\ \_ \right\rangle \cdot \left\langle fid_2,\ \_,\_,\ p_2,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle fid_2,\ pc_2,\ \_,\ c \cdot p_2,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |

Tabela 3.2.5: Instruções Get da SO

$P \vdash C_0 \to C_1$

| $\boldsymbol{fid_P[pc]}$ | **Cond** | $\boldsymbol{C_0}$ | $\boldsymbol{C_1}$ |
|---|---|---|---|
| `CGetL` $\boldsymbol{x}$ | | $\left\langle \_,\ pc,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ {}^*ts_0[\langle \boldsymbol{x},\ \texttt{globals} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ {}^*ts[\langle \boldsymbol{x},\ \texttt{local} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `CGetN` | | $\left\langle \_,\ pc,\ \_,\ c_1 : Str \cdot p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ {}^*ts_0[\langle c_1,\ \texttt{globals} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ c_1 : Str \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ {}^*ts[\langle c_1,\ \texttt{local} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `VGetL` $\boldsymbol{x}$ | | $\left\langle \_,\ pc,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ ts_0[\langle \boldsymbol{x}, \texttt{globals} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ ts[\langle \boldsymbol{x},\ \texttt{local} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `VGetG` $\boldsymbol{x}$ | | $\left\langle \_,\ pc,\ \_,\ p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ ts_0[\langle \boldsymbol{x}, \texttt{globals} \rangle] \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0$ |

Tabela 3.2.6: Instruções de Mutação da SO

$P \vdash C_0 \to C_1$

| $\boldsymbol{fid_P[pc]}$ | **Cond** | $\boldsymbol{C_0}$ | $\boldsymbol{C_1}$ |
|---|---|---|---|
| `SetL`$\boldsymbol{x}$ | | $\left\langle \_,\ pc,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0[\langle \boldsymbol{x},\ \texttt{globals} \rangle \twoheadrightarrow c]$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts[\langle \boldsymbol{x},\ \texttt{local} \rangle \twoheadrightarrow c],\ c \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `SetN` | | $\left\langle \_,\ pc,\ \_,\ c_1 \cdot c_2 : Str \cdot p,\ \_,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ c_1 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0[\langle c_2,\ \texttt{globals} \rangle \twoheadrightarrow c_1]$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ c_1 \cdot c_2 : Str \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts[\langle c_2,\ \texttt{local} \rangle \twoheadrightarrow c_1],\ c_1 \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |
| `BindL` $\boldsymbol{x}$ | | $\left\langle \_,\ pc,\ \_,\ v \cdot p,\ \_ \right\rangle \cdot \epsilon;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ v \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_;\ ts_0[\langle \boldsymbol{x},\ \texttt{globals} \rangle \mapsto v]$ |
| | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ v \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_$ | $\left\langle \_,\ pc+1,\ ts[\langle \boldsymbol{x},\ \texttt{local} \rangle \mapsto v],\ v \cdot p,\ \_,\ \_ \right\rangle \cdot A;\ \_$ |
| `UnsetL` $\boldsymbol{x}$ | $A = \epsilon$ | $\left\langle \_,\ pc,\ \_,\ p,\ \_ \right\rangle \cdot\ A;\ \_;\ ts_0$ | $\left\langle \_,\ pc+1,\ \_,\ p,\ \_ \right\rangle \cdot A;\ \_;\ ts_0 \setminus \langle \boldsymbol{x},\ \texttt{globals} \rangle$ |
| | | $\left\langle \_,\ pc,\ ts,\ p,\ \_ \right\rangle \cdot \epsilon;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ ts \setminus \langle \boldsymbol{x},\ \texttt{local} \rangle,\ p,\ \_ \right\rangle \cdot A;\ \_;\ \_$ |

Tabela 3.2.7: Intrução de chamada a funções da SO

$P \vdash C_0 \to C_1$

| $\boldsymbol{fid_P[pc]}$ | **Cond** | $\boldsymbol{C_0}$ | $\boldsymbol{C_1}$ |
|---|---|---|---|
| `FPushFuncD` ***int*** *nf* | | $\left\langle \_,\ pc,\ \_,\ \_,\ \_,\ pf \right\rangle \cdot A;\ \_;\ \_$ | $\left\langle \_,\ pc+1,\ \_,\ \_,\ \_,\ \langle \boldsymbol{int},\ nf \rangle \cdot pf \right\rangle \cdot\ A;\ \_;\ \_$ |
| `FCall` ***int*** | $\boldsymbol{int} = \lvert s1 \rvert \wedge$ $F[fid_2].ltparams = \alpha \wedge$ $getListaT(s1) = \alpha' \wedge$ $\lvert \alpha' \rvert = \lvert \alpha \rvert \ \wedge\ \alpha' <: \alpha$ $F[fid_2].lparams = \varkappa$ | $\left\langle fid_1,\ pc_1,\ ts_1,\ s1 \bullet p_1,\ \_,\ fid_2 \cdot pf_1 \right\rangle \cdot A;\ F;\ \_$ | $\left\langle fid_2,\ 0,\ ts_2[\varkappa_0, \ldots, \varkappa_{\lvert \varkappa \rvert} \to s1],\ \epsilon,\ \epsilon,\ \epsilon \right\rangle$ $\cdot \left\langle fid_1,\ pc_1+1,\ ts_1,\ p_1,\ \_,\ pf_1 \right\rangle \cdot A;\ F;\ \_$ |

Tabela 3.2.8: Instruções de arrays da SO

$P \vdash C_0 \rightarrow C_1$

| $\boldsymbol{fid_P[pc]}$ | **Cond** | $\boldsymbol{C_0}$ | $\boldsymbol{C_1}$ |
|---|---|---|---|
| `BaseL` $\boldsymbol{x}$ | $A \neq \epsilon$ | $\left\langle \_,\ pc,\ \_,\ \_,\ rbm,\ \_ \right\rangle \cdot A;\ F$ | $\left\langle \_,\ pc+1,\ \_,\ \_,\ \langle \boldsymbol{x},\ \texttt{local} \rangle,\ \_ \right\rangle \cdot A;\ \_$ |
| `BaseGC` ***int*** | $\lvert s1 \rvert = \boldsymbol{int} \wedge$ $c \notin TvEU \wedge$ $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ s1 \bullet c : \texttt{Str} \cdot p,\ rbm,\ \_ \right\rangle \cdot A;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ p,\ \langle c,\ \texttt{globals} \rangle,\ \_ \right\rangle \cdot A;\ \_$ |
| | $\lvert s1 \rvert = \boldsymbol{int} \wedge$ $c \in TvEU \wedge$ $A \neq \epsilon$ | $\left\langle \_,\ pc,\ ts,\ s1 \bullet c : \texttt{Str} \cdot p,\ rbm,\ \_ \right\rangle \cdot A;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ p,\ \langle null,\ c \rangle,\ \_ \right\rangle \cdot A;\ \_$ |
| `QueryM` ***int*** $\boldsymbol{\rho}$ $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $\boldsymbol{\rho} = CGet \wedge$ $(\kappa = ET \vee$ $\kappa = EI) \wedge$ $\lvert s1 \rvert = \boldsymbol{int}$ | $\left\langle \_,\ pc,\ ts,\ s1 \bullet p,\ \langle \boldsymbol{x}_1,\ tvar_1 \rangle_{varId1},\ \_ \right\rangle \cdot A;\ \_$ | $\left\langle \_,\ pc+1,\ ts,\ {}^*ts[varId_1][\boldsymbol{x}] \cdot p,\ \epsilon,\ \_ \right\rangle \cdot A;\ \_$ |
| `SetM` $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $(\kappa = ET \vee$ $\kappa = EI) \wedge$ $\lvert s1 \rvert = \boldsymbol{int}$ | $\left\langle \_,\ pc,\ ts,\ c_1 \cdot s1 \bullet p,\ ts,\ \langle \boldsymbol{x}_1,\ tvar_1 \rangle_{varId1},\ \_ \right\rangle \cdot A;\ \_$ | $\left\langle \_,\ pc+1,\ ts[varId_1 \twoheadrightarrow^* ts[varId_1][\boldsymbol{x} \twoheadrightarrow c_1]],\ c \cdot p,\ \epsilon,\ \_ \right\rangle \cdot A;\ \_$ |

## 3.3 Semântica Estática

A semântica estática (SE) apresentada nesta secção determina se pode ser atribuído um tipo válido a um programa HHAS. Caso isso aconteça então esse programa não é vulnerável a ataques SQLI ou XSS.

Na Figura 3.6 é apresentado o ambiente considerado na análise de tipos. O ambiente $\Gamma$ é um mapa parcial de identificadores de funções para a sua respetiva definição e tipo. Consideramos que as interfaces das funções, *built-in* do PHP, de bibliotecas externas, e de *sensitive sink*, são sempre bem definidas e são representadas no ambiente apenas pelo seu tipo. Também consideramos que a sanitização de um valor torna-o sanitizado para todas as *sensitive sink* e, portanto, temos apenas um tipo `Untainted` e um tipo `Tainted`.

$$\Gamma^F : \mathit{FID} \rightharpoonup \langle \text{Instrução}^+,\ \mathit{Função} \rangle$$
$$\Gamma^I : \mathit{FID} \rightharpoonup \langle \mathit{Função} \rangle$$

$$\Gamma = \Gamma^F \cup \Gamma^I$$

Figura 3.6: Formato do ambiente de um programa HHAS

As conversões apresentadas na Tabela 3.3.1 permitem converter um *type hint* da linguagem HHAS para um tipo mais geral, utilizado na análise, que tem apenas em consideração o quão malicioso é um tipo.

Tabela 3.3.1: Regras para converter um *type hint* da linguagem HHAS para um tipo mais geral `T`

| ***Type hint*** | **Tipo `T` resultante** |
|---|---|
| *sstring* \| *integer* \| *boolean* \| *untainted* | `Untainted` |
| *string* \| *tainted* | `Tainted` |
| *array* | `TaintedArray` $\epsilon$ |
| sem *type hint* definido | `Tainted` |

As principais regras de tipos são apresentadas na Figura 3.7. A regra de tipos **Programa BT**, verifica se um programa é bem tipificado. Nesta regra a função *CDom* retorna a lista dos tipos das funções do ambiente $\Gamma$, que é ordenada alfabeticamente pelo nome da respetiva função. O pré-requisito da lista ser ordenada, irá ser apresentada na Secção 3.3.5, garante que a análise de tipos é determinística.

A regra de tipos **Calc. Ponto Fixo Inf. Ambiente**, segue um algoritmo do cálculo do ponto fixo numa inferência consecutiva de novos ambientes. Esta regra será explicada em mais detalhe na Secção 3.3.5. A segunda e a terceira linha desta regra exigem que todas as funções da lista *lf* sejam bem tipificadas no ambiente $\Gamma$ recorrendo à regra de tipos **Lista de Funções BT**. A regra de tipos **Lista de**

**Funções BT**, permite validar consecutivamente que a função $f$ e as funções em $lf$ são bem tipificadas, no ambiente $\Gamma$, recorrendo à regra de tipos **Função BT**. Em cada uma destas validações pode ser feita a inferência de um novo ambiente $\Gamma'$ que é considerado na análise da função seguinte. Na primeira linha desta regra é possível observar que é inferido o ambiente $\Gamma''$ que corresponde ao ambiente inferido na validação da última função da lista $lf$.

A regra de inferência **Função BT** verifica se uma função é bem tipificada e não é vulnerável num determinado ambiente. Nesta regra, $B$ é um mapa de Endereços para mapas de Etiquetas para *Frame Branch* que será apresentado em mais detalhe na Secção 3.3.2. Uma *Frame Branch* é definida pelos tipos apresentados na segunda linha dessa regra. Inicialmente, nas funcionalidades mais simples são considerados apenas alguns dos tipos de uma *Frame Branch*. Os restantes tipos, $pf$, $ta$, $u$ e $adt$ irão ser apresentados mais à frente. Uma *Frame Branch* é composta por:

- $i$: o endereço da instrução.
- $TS$: um mapa de endereços para funções que mapeiam `VarID` para `TCaixa`, onde a notação $TS_i[\varphi]$ indica o tipo da variável $\varphi$ na linha $i$.
- $P$: um mapa de endereços para pilhas de tipos `THH`, onde $P_i$ é o tipo da pilha na linha $i$.

Se for encontrado um tipo $b$ e $B$ de forma a que $\Gamma,\ b,\ B \vdash C : f$ então significa que o verificador aceita o código da função $f$. A regra de tipos **Código Função** verifica o corpo de uma função. O julgamento $\Gamma,\ b,\ B \vdash C : f$ significa que, dado um ambiente $\Gamma$ e a informação dos tipos $b$ e $B$, a execução do *array* de instruções $C$ não causa um erro de tipos e é consistente com o tipo $f$. Nessa regra a lista de tipos $\alpha$ é convertida pela função $deTHintParaT$, através das regras de conversão apresentadas na Tabela 3.3.1, para uma lista de tipos `T`. O mapa $TS_1$ guarda todos os tipos de cada um dos parâmetros. O tipo $b_1$ define o tipo da *Frame Branch* corrente na primeira instrução.

Da Secção 3.3.1 à 3.3.7 são apresentadas as regras de tipos das instruções. Em cada Secção é apresentada uma funcionalidade considerada na análise. Inicialmente são apresentadas as funcionalidades mais simples e consoante vão sendo apresentadas mais funcionalidades as regras de tipos de uma instrução podem ser apresentadas novamente contendo mais restrições por serem consideradas mais funcionalidades e mais tipos. Estas regras de tipos descrevem um conjunto de restrições entre tipos de variáveis, espaços na pilha e relações de *aliases* em diferentes instantes do programa. As regras de tipos das instruções são apresentadas com o mesmo formato da semântica operacional, mas em vez de ser utilizada uma máquina de estados são feitos juízos, onde as primeiras colunas apresentam as condições na linha corrente e a última coluna apresenta as restrições de tipos no sucessor da linha corrente.

Na semântica estática para além das notações utilizadas na semântica operacional são utilizadas outras notações.

A notação ${}^{*}TS_i[\varphi]$ permite obter o tipo `T`, contido no tipo `TCaixa`, da variável que tem $\varphi$ como `VarID`. A notação $TS_i[\varphi \twoheadrightarrow \tau]$ indica que o tipo `TCaixa` da variável $\varphi$ mantém-se, alterando apenas o seu conteúdo, no caso da variável ainda não estiver definida, o tipo dessa variável é um novo tipo `TCaixa` que contem o tipo $\tau$.

A função *Dom* retorna o domínio de um tipo, por exemplo, $Dom(TS_i)$ retorna o domínio da tabela de símbolos na instrução $i$ que corresponde um conjunto de `VarID`.

$$\frac{\begin{array}{c}\Gamma \vdash lf \\ lf = CDom(\Gamma^F) \\ CDom(\Gamma^F) \neq \emptyset \\ \Gamma = \Gamma^F \cup \Gamma^I\end{array}}{\Gamma \vdash bt} \text{ (Programa BT)}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma' \qquad \Gamma' \vdash bt \\ \Gamma = \Gamma' \vee \Gamma \neq \Gamma' \\ lf',\ \Gamma \vdash f \dashv \Gamma' \\ (lf = f \cdot lf')\end{array}}{\Gamma \vdash lf} \text{ (Calc. Ponto Fixo Inf. Ambiente)}$$

$$\frac{\begin{array}{c}\Gamma'' = \Gamma' \\ lf',\ \Gamma' \vdash f' \qquad lf = \epsilon \\ (lf = f' \cdot lf') \vee (lf = \epsilon)\end{array}}{lf,\ \Gamma \vdash f \dashv \Gamma''} \text{ (Lista de Funções BT)}$$

$$\frac{\begin{array}{c}\Gamma[fid] = \langle C,\ f\rangle \\ b = \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle \\ \Gamma,\ b,\ B \vdash C : f \dashv \Gamma' \\ f = \langle fid,\ ltparams,\ lparams,\ lparamsPRef,\ ret,\ tipoExt\rangle\end{array}}{\Gamma \vdash f \dashv \Gamma'} \text{ (Função BT)}$$

$$\frac{\begin{array}{c}TS_0 = \epsilon \\ \varsigma = deTHintParaT(\alpha) \\ TS_1 = TS_0[\varkappa_0, \ldots, \varkappa_{|\varkappa|} \twoheadrightarrow \varsigma] \\ P_1 = \epsilon \\ b_1 = \langle 1, TS_1,\ P_1,\ \epsilon,\ \epsilon,\ \epsilon,\ \epsilon,\ 0\rangle_b \\ B_1 = \{\epsilon\} \\ f = \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle \\ \forall i \in Dom(C).\ \Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : f\end{array}}{\Gamma,\ b,\ B \vdash C : f \dashv \Gamma'} \text{ (Código Função)}$$

Figura 3.7: Principais regras de tipos

### 3.3.1 Instruções Base

Da Tabela 3.3.2 à Tabela 3.3.7 são apresentadas as regras de tipos das instruções simples que não consideram a análise de *arrays*, variáveis globais, entradas de utilizador, *aliases* e variáveis acedidas dinamicamente. Apesar de não serem consideradas variáveis globais, as variáveis locais são consideradas variáveis globais no contexto *main*.

Nesta Secção é utilizada a função $ObtT$ que obtém o tipo `T` de um tipo `THH` ou de um tipo `TCaixa`.

Tabela 3.3.2: Funcionalidades base nas instruções Básicas da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `Nop` | | $i \in Dom(C)$ |
| `PopC`<br>PopV | $\Gamma \vdash P_i = \mu \cdot \beta$ | $\Gamma \vdash P_{i+1} = \beta$<br>$i+1 \in Dom(C)$ |
| `Dup` | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau \in$ `TP` | $\Gamma \vdash P_{i+1} = \tau \cdot \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.3: Funcionalidades base nas instruções de Literais da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `Null`<br>`True`<br>`Int int` | | $\Gamma \vdash P_{i+1} =$ `Untainted` $\cdot P_i$<br>$i+1 \in Dom(C)$ |
| `String s` | | $\Gamma \vdash P_{i+1} =$ (`Untainted`, `s`) $\cdot P_i$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.4: Funcionalidades base nas instruções de Operadores da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `Add` | $\Gamma \vdash P_i = \mu \cdot \mu' \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau' = ObtT(\mu')$<br>$\tau,\ \tau' \in$ `TP` | $\Gamma \vdash P_{i+1} =$ `Untainted` $\cdot \beta$<br>$i+1 \in Dom(C)$ |
| `Abs` | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau \in$ `TP` | $\Gamma \vdash P_{i+1} =$ `Untainted` $\cdot \beta$<br>$i+1 \in Dom(C)$ |
| `BitNot` | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau \in$ `TP` | $\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.4: Funcionalidades base nas instruções de Operadores da SE (cont.)

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `Concat` | $\Gamma \vdash P_i = \mu \cdot \mu' \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau' = ObtT(\mu')$<br>$\tau = \texttt{Untainted} \wedge \tau' = \texttt{Untainted}$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |
| | $\Gamma \vdash P_i = \mu \cdot \mu' \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau' = ObtT(\mu')$<br>$\tau = \texttt{Tainted} \vee \tau' = \texttt{Tainted}$ | $\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |
| `CastBool` | $\Gamma \vdash P_i = \mu \cdot \beta$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |
| `CastString` | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau \in \texttt{TP}$ | $\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |
| `Print` | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau = \texttt{Untainted}$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.5: Funcionalidades base nas instruções Isset, Empty, and type querying da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `IssetC`<br>`IsTypeC` $s$ | $\Gamma \vdash P_i = \mu \cdot \beta$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |
| `IssetL` $x$<br>`IsTypeL` $x$ $s$ | | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.6: Funcionalidades base nas instruções Get da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `CGetL` $x$ | $tf = \texttt{main}$<br>$\varphi = (x,\ \texttt{globals})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (^* TS_i[\varphi],\ \varphi) \cdot P_i$<br>$i+1 \in Dom(C)$ |
| | $tf = \texttt{main}$<br>$\varphi = (x,\ \texttt{globals})$<br>$\varphi \notin Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (\texttt{Tainted},\ \varphi) \cdot P_i$<br>$i+1 \in Dom(C)$ |
| | $tf = \texttt{função}$<br>$\varphi = (x,\ \texttt{local})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (^* TS_i[\varphi],\ \varphi) \cdot P_i$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.7: Funcionalidades base nas instruções de Mutação da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `SetL` $x$ | $tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\varphi = (x,\ \texttt{globals})$ | $\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \tau]$<br>$\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.7: Funcionalidades base nas instruções de Mutação da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \asymp,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | sub-restrições na linha i | restrições no sucessor de i |
|---|---|---|---|
| SetOpL $x\ \omega$ | $\omega \neq Concat\ \wedge\ tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\varphi = (x,\ \texttt{globals})$ | | $\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \texttt{Untainted}]$<br>$\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |
| | $\omega = Concat\ \wedge\ tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\varphi = (x,\ \texttt{globals})$ | $\tau = ObtT(\mu)$<br>$\varphi \in Dom(TS_i)$<br>$\tau' = {}^*TS_i[\varphi]$<br>$\tau = \texttt{Untainted} \wedge \tau' = \texttt{Untainted}$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |
| | | $\tau = ObtT(\mu)$<br>$\varphi \in Dom(TS_i)$<br>$\tau' = {}^*TS_i[\varphi]$<br>$\tau = \texttt{Tainted} \vee \tau' = \texttt{Tainted}$ | $\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \texttt{Tainted}]$<br>$\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.7: Funcionalidades base nas instruções de Mutação da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \asymp,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| IncDecL $x\ \delta$ | $tf = \texttt{main}$<br>$\varphi = (x,\ \texttt{globals})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = {}^*TS_{i+1}[\varphi] \cdot P_i$<br>$i+1\ \in\ Dom(C)$ |
| | $tf = \texttt{main}$<br>$\varphi = (x,\ \texttt{globals})$<br>$\varphi \notin Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot P_i$<br>$i+1\ \in\ Dom(C)$ |

### 3.3.2 Fluxo de Controle

Os programas HHAS têm tipos dinâmicos, o que significa que o tipo de uma variável pode variar durante a execução do programa. Isto significa que uma variável pode ter tipos diferentes consoante o fluxo do programa que considerar.

Por este motivo, para garantir que não existe nenhuma combinação de fluxo do programa que torne o programa vulnerável, são consideradas as instruções de salto para analisar as condições e ciclos dos programas. Inicialmente são guardados num mapa as etiquetas com o formato $L$ **int** do programa HHAS que mapeiam a respetiva linha do programa. Isto permite atualizar a linha que está a ser analisada quando é encontrada uma instrução de salto. Outro mapa liga etiquetas a cópias do estado da análise denominados de Frame Branches.

Numa condição são sempre gerados dois *Frame Branches* uma para analisar o caminho no programa em que a condição é verdadeira e outro para analisar o caminho em que a condição é falsa. Após serem feitas as análises desta condição existem dois *Frame Branches* que podem guardar diferentes informações de tipos da análise e, portanto, a análise poderia prosseguir de duas formas: ou são feitas duas análises diferentes considerando cada um dos diferentes *Frame Branches* ou é feita uma junção dos dois *Frame Branches* que tem de considerar o pior caso possível entre cada uma das informações de tipos. Como esta análise está a ser feita para analisar programas que exigem rápidos ciclos de desenvolvimento a abordagem escolhida faz a junção dos dois *Frame Branches*.

Nesta análise de tipos cada uma das instruções é analisada de uma forma independente das outras, exceto a instrução `Jmp`. Na análise desta instrução é tido por base o modelo de compilação para HHAS e, portanto, como esta instrução é sempre seguida de uma etiqueta $l$, é utilizada esta característica para fazer sua análise.

A análise das instruções de salto é feita da seguinte forma:

- Na análise de um salto para a frente é feita uma cópia do estado da análise corrente onde é atualizada a linha corrente para a linha onde é feito o salto. Esta cópia é guardada no mapa de etiquetas para *Frame Branches* (no mapa $B_i$). Isto permite que a análise corrente prossiga para a instrução seguinte. Quando a análise corrente encontrar uma destas etiquetas tem de consultar o mapa referido anteriormente. Caso tenha alguma cópia nessa etiqueta significa que é necessário proceder à junção de *Frame Branches*, que é apresentada mais abaixo.

- No caso de o salto ser condicional, mas ser feito para trás significa que estamos perante um ciclo e nesse caso é feito o mesmo procedimento que no ponto acima e o ciclo é analisado iterativamente. Este algoritmo assemelha-se ao cálculo de pontos fixos. A convergência é garantida por sabermos que não existe um

número infinito de variáveis e porque em cada ciclo as variáveis só podem alterar o seu tipo para um tipo mais malicioso como é representado na Tabela 3.3.8

Tabela 3.3.8: Regras para a junção de dois tipos de uma variável na junção de duas *Frame Branch*

| **Tipo numa *Frame Branch*** | **Tipo na outra *Frame Branch*** | **Resultado da junção** |
|---|---|---|
| $\texttt{Untainted}_1$ | $\texttt{Untainted}_2$ | $\texttt{Untainted}_\tau$ |
| $\texttt{Untainted}_1$ | $\texttt{UntaintedArray}_2$ | |
| $\texttt{Tainted}_1$ | Qualquer tipo | $\texttt{Tainted}_\tau$ |
| Variável não definida | Qualquer tipo | |
| $\texttt{UntaintedArray}_1$ | $\texttt{UntaintedArray}_2$ | $\texttt{UntaintedArray}_\tau$ |
| $\texttt{Untainted}_1$ | $\texttt{TaintedArray}_2\ \texttt{Array}_2$ | $\texttt{TaintedArray}_2\ \texttt{Array}_2$ |
| $\texttt{UntaintedArray}_1$ | $\texttt{TaintedArray}_2\ \texttt{Array}_2$ | |
| $\texttt{TaintedArray}_1\ \texttt{Array}_1$ | $\texttt{TaintedArray}_2\ \texttt{Array}_2$ | $\texttt{TaintedArray}_\tau\ \texttt{Array}_\varpi$ |

Nesta Secção são utilizadas as funções:

- *linhaEtiqueta* recebe uma etiqueta e retorna o número da linha dessa etiqueta.
- *etiquetaLinha* recebe o número de uma linha e retorna a etiqueta que está nessa linha.
- *clonaFrameBranch* trata de clonar uma *Frame Branch*.
- *juntaFrameBranches* faz a junção de dois *Frame Branches*. Nesta junção são comparadas as duas tabelas de símbolos e para cada variável é atribuído o tipo mais malicioso possível que resulta da junção de dois tipos.

A junção de dois tipos é apresentada na Tabela 3.3.8, onde nas duas primeiras colunas é apresentado o tipo de uma variável e na terceira é apresentado o tipo que resulta da junção dos tipos das duas primeiras colunas.

Na Tabela 3.3.9 são apresentadas as instruções de fluxo de controle à exceção da instrução de retorno que é apresentada na secção seguinte.

Tabela 3.3.9: Funcionalidades nas instruções de Fluxo de Controle da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \nsim,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `Jmp` $l$ | $linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} > i \wedge l \in Dom(B_i)$<br>$etiquetaLinha(i+1) = l'$<br>$juntaFrameBranches(B[l], b_i) = b_{\boldsymbol{x}}$ | $\Gamma \vdash b_{i+1} = B[l']$<br>$\Gamma \vdash B_{i+1} = B_i[l \mapsto b_{\boldsymbol{x}}]$<br>$i+1 \in Dom(C)$ |
| | $linhaEtiqueta(l) > i \wedge l \notin Dom(B_i)$<br>$etiquetaLinha(i+1) = l'$ | $\Gamma \vdash b_{i+1} = B[l']$<br>$\Gamma \vdash B_{i+1} = B_i[l \mapsto b_i]$<br>$i+1 \in Dom(C)$ |
| $l$ | $l \notin Dom(B_i)$ | $i+1 \in Dom(C)$ |
| | $l \in Dom(B_i)$ | $\Gamma \vdash b_{i+1} = juntaFrameBranches(B[l], b_i)$<br>$\Gamma \vdash B_{i+1} = B_i \setminus l$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.9: Funcionalidades nas instruções de Fluxo de Controle da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \nsim,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `JmpZ` $l$ | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} > i \wedge l \notin Dom(B_i)$<br>$clonaFrameBranch(b_i) = b_{\boldsymbol{x}}$ | $\Gamma \vdash P_{i+1} = \beta$<br>$\Gamma \vdash B_{i+1} = B_i[l \mapsto b_{\boldsymbol{x}}]$<br>$i+1 \in Dom(C)$ |
| | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} > i \wedge l \in Dom(B_i)$<br>$juntaFrameBranches(B[l], b_i) = b_{\boldsymbol{x}}$ | $\Gamma \vdash P_{i+1} = \beta$<br>$\Gamma \vdash B_{i+1} = B_i[l \mapsto b_{\boldsymbol{x}}]$<br>$i+1 \in Dom(C)$ |
| | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} < i \wedge l \notin Dom(B_i)$<br>$clonaFrameBranch(b_i) = b_{\boldsymbol{x}}$ | $\Gamma \vdash P_{\boldsymbol{x}} = \beta$<br>$\Gamma \vdash B_{\boldsymbol{x}} = B_i[l \mapsto b_{\boldsymbol{x}}]$<br>$\boldsymbol{x} \in Dom(C)$ |
| | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} < i \wedge l \in Dom(B_i)$<br>$\Gamma \vdash b_i \neq B_i[l]$<br>$juntaFrameBranches(B[l], b_i) = b_{\boldsymbol{x}}$ | $\Gamma \vdash P_{\boldsymbol{x}} = \beta$<br>$\Gamma \vdash B_{\boldsymbol{x}} = B_i[l \mapsto b_{\boldsymbol{x}}]$<br>$\boldsymbol{x} \in Dom(C)$ |
| | $\Gamma \vdash P_i = \mu \cdot \beta$<br>$linhaEtiqueta(l) = \boldsymbol{x} \wedge \boldsymbol{x} < i \wedge l \in Dom(B_i)$<br>$\Gamma \vdash b_i = B_i[l]$ | $\Gamma \vdash P_{i+1} = \beta$<br>$\Gamma \vdash B_{i+1} = B_i \setminus l$<br>$i+1 \in Dom(C)$ |

### 3.3.3 Função

O *type hint* de cada parâmetro e do retorno de cada função é guardado pelo tipo *Função*.

Caso a função seja uma função *built-in* do PHP, uma função de uma biblioteca externa ou uma função de sensitive sink, o *type hint* dos parâmetros e o *type hint* de retorno dessa função devem estar definidos num ficheiro de configuração que é utilizado pelo analisador de tipos para definir as restrições de tipos dessas funções, na Listagem 3.3 está representado um exemplo possível deste ficheiro.

No caso da função ser definida pelo utilizador, cada parâmetro deve ter o seu *type hint* definido, caso não o tenha, o seu *type hint* é considerado `Tainted`. Na Tabela 3.3.10 na instrução de retorno é verificado se o tipo retornado corresponde ao tipo esperado pelo *type hint* do retorno da respectiva função. Caso a função não tenha definido o *type hint* do retorno, a função passa a ter como tipo de retorno o tipo inferido pela análise. Como podemos ver na regra de tipos da instrução

`RetC` o estado do ambiente é alterado onde se passa a conhecer o tipo de retorno da função *FID*. As instruções de chamada a uma função, apresentadas na Tabela 3.3.11 utilizam o tipo da função chamada para atualizar o estado da análise. Nesta instrução é utilizado o mapa *pf* de Endereços para pilhas de *FID* e a função *getListaT* que retira o `T` contido num `THH` para cada elemento de uma lista de tipos `THH` e cria uma lista de tipos `T`, que são os tipos dos parâmetros da chamada à função. As regras de subtipos definidas na Figura 3.8 permitem verificar se o tipo passado por parâmetro tem o tipo esperado.

```
1
2  //NonSensitive
3  .function <"HH\\int" "HH\\int" hh_type extended_hint >
       strlen(<"HH\\string" "HH\\string"  > $param1)
4
5
6  .function <"HH\\string" "HH\\string" hh_type extended_hint >
       stristr(<"HH\\string" "HH\\string"  > $param1,<"HH\\string"
       "HH\\string"  > $param2)
7
8  .function <"HH\\string" "HH\\string" hh_type extended_hint >
       http_build_query(<"HH\\string" "HH\\string"  $param1)
9
10 .function <"HH\\int" "HH\\int" hh_type extended_hint >
       count($param1)
11
12 //Sanitisation
13 .function <"HH\\sstring" "HH\\sstring" hh_type extended_hint >
       htmlentities(<"HH\\string" "HH\\string"  > $param1,
       <"HH\\string" "HH\\string"> $param2 = DV1("""NULL"""))
14
15 .function <"HH\\sstring" "HH\\sstring" hh_type extended_hint >
       mysql_real_escape_string(<"HH\\string" "HH\\string"  >
       $param1)
16
17 .function <"HH\\sstring" "HH\\sstring" hh_type extended_hint >
       urlencode(<"HH\\string" "HH\\string"  > $param1)
18
19 //Sensitive sinks
20 .function <"HH\\tainted" "HH\\tainted" hh_type extended_hint >
       mysql_query(<"HH\\sstring" "HH\\sstring"  > $param1)
```

Listagem 3.3: Exemplo de ficheiro de configuração onde é definida a interface das funções built-in

$$\frac{}{\vdash \texttt{Untainted} <: \eta} \qquad \frac{}{\vdash \texttt{UntaintedArray} <: \eta}$$

$$\frac{\eta \neq \texttt{sstring} \wedge \eta \neq \texttt{untainted}}{\vdash \texttt{Tainted} <: \eta} \qquad \frac{}{\vdash (\texttt{TaintedArray}\ \varpi) <: \texttt{array}}$$

$$\frac{\forall i \in [0..n-1]\ \Gamma \vdash listaT[i] <: parametros[i]}{\Gamma \vdash listaT : (\tau_1 \cdot \tau_2 \cdots \tau_n) <: parametros : \alpha}$$

Figura 3.8: Regras de subtipos

Tabela 3.3.10: Funcionalidade de chamada a funções na instrução de Retorno da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| RetC | $tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$ | |
| | $tf = \texttt{função} \wedge \gamma \neq \epsilon$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau <: \gamma$ | |
| | $tf = \texttt{função} \wedge \gamma = \epsilon$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$ | $f' = \langle FID,\ \alpha,\ \varkappa,\ \kappa,\ \tau,\ \epsilon\rangle$<br>$\Gamma' = \Gamma[FID \mapsto \langle C,\ f'\rangle]$<br>$\dashv \Gamma'$ |

Tabela 3.3.11: Funcionalidade e instruções de chamada a funções da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| FPushFuncD ***int*** *nf* | | $\Gamma \vdash pf_{i+1} = \{|\texttt{função},\ nf|\} \cdot pf_i$<br>$i+1 \in Dom(C)$ |
| FCall ***int*** | $\Gamma \vdash pf_i = fid' \cdot pf'$<br>$\Gamma[fid'].\texttt{tipo} = \langle fid',\ \alpha',\ \varkappa',\ \kappa',\ \gamma',\ e'\rangle$<br>$\Gamma \vdash P_i = \beta' \bullet \beta$<br>$\boldsymbol{int} = \lvert\beta'\rvert \wedge \varsigma = getListaT(\beta')$<br>$\varsigma <: \alpha'$ | $\Gamma \vdash P_{i+1} = \gamma' \cdot \beta$<br>$\Gamma \vdash pf_{i+1} = pf'$<br>$i+1 \in Dom(C)$ |

### 3.3.4 *Arrays* associativos

A ferramenta consegue analisar *arrays* considerando que tem os valores indexados como se fosse um *array* associativo. Em PHP e Hack um objeto pode ser tratado como um *array* e, portanto, é muito mais comum aceder estaticamente a uma posição constante do *array*. Apesar da ferramenta não considerar objetos na sua análise, esta característica, referida anteriormente, foi tida em conta tratando um *array* como um mapa que pode ter um tipo diferente em cada posição. Isto significa que num *array* é possível aceder estaticamente a uma posição com o tipo `Untainted`, mesmo que todas as outras posições desse *array* sejam do tipo `Tainted`.

O programa da Listagem 3.4, mostra um exemplo de um *array* que tem uma posição com um tipo `Untainted` e outra com o tipo `Tainted`. Este programa é válido porque a posição do *array* utilizada para a concatenação de `Strings`, na linha 9, é a posição com o tipo `Untainted` e, portanto, o resultado dessas concatenações dá origem a um tipo `Untainted` que é o tipo esperado pela função de *sensitive sink* `echo`.

As regras de tipos das instruções de *arrays* são representadas na Tabela 3.3.12.

```
1  <?php
2
3      confirmaPedidoMenu1();
4
5      function confirmaPedidoMenu1(){
6          $menu1["pratoPrincipal"] = "Espetada da Madeira";
7          $menu1["bebida"] = $_POST["bebidaEscolhida"];
8
9          echo "O pedido do Menu 1,". $menu1["pratoPrincipal"] ."
               foi feito com sucesso";
10     }
11 ?>
```

Listagem 3.4: Exemplo de um *array* do tipo `Untainted TArray` que tem um elemento `Untainted` e outro `Tainted`

Tabela 3.3.12: Funcionalidade e instruções de *arrays* da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `Array` @A_0 | $tf =$ `main` | $\Gamma \vdash P_{i+1} =$ `UntaintedArray` $\cdot P_i$<br>$i+1 \in Dom(C)$ |
| `BaseL` $\boldsymbol{x}$ | $tf =$ `main`<br>$\Gamma \vdash rbm_i = \epsilon$ | $\Gamma \vdash rbm_{i+1} = (\boldsymbol{x},$ `globals`$)$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.12: Funcionalidade e instruções de *arrays* da SE (cont.)

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| ***C[i]*** | **restrições na linha i** | **sub-restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|---|
| `QueryM` ***int*** $\boldsymbol{\rho}$ $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $\boldsymbol{\rho} =$ `CGet` $\wedge$ $tf =$ `main` $\wedge$ $(\kappa =$ `ET` $\vee$ $\kappa =$ `EI`$)$ $\lvert\beta^{'}\rvert =$ ***int*** $\Gamma \vdash P_i = \beta^{'} \cdot \beta$ $\Gamma \vdash rbm = \varphi$ | $\varphi \in Dom(TS_i)\ \wedge\ \tau =\ ^*TS_i[\varphi]$ $\tau \in$ `TArray` $\wedge\ \tau =$ `UntaintedArray` | $\Gamma \vdash P_{i+1} =$ (`Untainted` $\varphi) \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)\ \wedge\ \tau =\ ^*TS_i[\varphi]$ $\tau \in$ `TArray` $\wedge\ \tau =$ (`TaintedArray` $\varpi$) $\mathbf{x} \in Dom(\varpi)\ \tau^{'} = \varpi[\mathbf{x}]$ | $\Gamma \vdash P_{i+1} = (\tau^{'}\ \varphi) \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)\ \wedge\ \tau =\ ^*TS_i[\varphi]$ $\tau \in$ `TArray` $\wedge\ \tau =$ (`TaintedArray` $\varpi$) $\mathbf{x} \notin Dom(\varpi)$ | $\Gamma \vdash P_{i+1} =$ (`Tainted` $\varphi) \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)\ \wedge\ \tau =\ ^*TS_i[\varphi]$ $\tau \notin$ `TArray` | $\Gamma \vdash P_{i+1} = (\tau\ \varphi) \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \notin Dom(TS_i)$ | $\Gamma \vdash P_{i+1} =$ (`Tainted` $\varphi) \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | $\boldsymbol{\rho} =$ `Isset` $\vee\ \boldsymbol{\rho} =$ `Empty` $\lvert\beta^{'}\rvert =$ ***int*** $\Gamma \vdash P_i = \beta^{'} \cdot \beta$ $\Gamma \vdash rbm = \varphi$ | | $\Gamma \vdash P_{i+1} =$ `Untainted` $\cdot\ \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |

Tabela 3.3.12: Funcionalidade e instruções de *arrays* da SE (cont.)

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| ***C[i]*** | **restrições na linha i** | **sub-restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|---|
| `SetM` ***int*** $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $tf =$ `main` $(\kappa =$ `ET` $\vee\ \kappa =$ `EI`$)$ $\lvert\beta^{'}\rvert =$ ***int*** $\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$ $\Gamma \vdash rbm = \varphi$ | $\varphi \in Dom(TS_i)$ $\tau =\ ^*TS_i[\varphi] \wedge \tau \in$ TaintedArray $\tau^{'} = ObtT(\mu) \wedge \tau^{'} =$ `Untainted` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\Gamma \vdash TS_{i+1} = TS_i[\varphi \mapsto \tau[x \twoheadrightarrow \tau^{'}]]$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)$ $\tau =\ ^*TS_i[\varphi] \wedge \tau =$ `UntaintedArray` $\tau^{'} = ObtT(\mu) \wedge \tau^{'} =$ `Untainted` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)$ $\tau =\ ^*TS_i[\varphi] \wedge \tau =$ `UntaintedArray` $\tau^{'} = ObtT(\mu) \wedge \tau^{'} =$ `Tainted` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\Gamma \vdash TS_{i+1} = TS_i[\varphi \mapsto$ (`TaintedArray` $\epsilon)]$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)$ $\tau =\ ^*TS_i[\varphi] \wedge \tau \in$ TaintedArray $\tau^{'} = ObtT(\mu) \wedge \tau^{'} =$ `Tainted` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\Gamma \vdash TS_{i+1} = TS_i[\varphi \mapsto \tau[x \twoheadrightarrow \tau^{'}]$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \in Dom(TS_i)$ $\tau =\ ^*TS_i[\varphi] \wedge \tau \in$ `TP` $\tau^{'} = ObtT(\mu) \wedge \tau^{'} \in$ `TP` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\tau^{'} =$ `Tainted` $\Rightarrow \Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \tau^{'}]$ $\tau^{'} =$ `Untainted` $\Rightarrow \Gamma \vdash TS_{i+1} = TS_i$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |
| | | $\varphi \notin Dom(TS_i)\ \wedge\ \tau^{'} = ObtT(\mu)$ $\tau^{'} \in$ `TP` | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$ $\tau =$ (`TaintedArray` $\epsilon)$ $\Gamma \vdash TS_{i+1} = TS_i[\varphi \mapsto \tau[x \twoheadrightarrow \tau^{'}]$ $\Gamma \vdash rbm_{i+1} = \epsilon$ $i+1\ \in\ Dom(C)$ |

### 3.3.5 Entradas de utilizador e variáveis do exterior

As variáveis "*super* globais" (*superglobals*) permitem aceder tanto a variáveis de entrada de utilizador como a variáveis globais. Estas variáveis são *arrays* pré-definidos

em PHP e, portanto, são acedidas em HHAS através das instruções de *arrays* que são apresentadas na Tabela 3.3.17.

As atribuições feitas às variáveis de entrada de utilizador são ignoradas e, portanto, estas variáveis têm sempre o tipo `Tainted`.

Por se tratar de uma linguagem dinamicamente tipificada, os tipos das variáveis globais não são definidos. Na nossa análise foi utilizada uma abordagem bastante conservadora, que considera que o tipo destas variáveis não é conhecido à entrada da função e, portanto, é considerado que têm o tipo `Tainted`. Dando assim origem a falsos positivos, mas prevenindo a origem de falsos negativos. O programa da Listagem 3.5 é considerado vulnerável por esse motivo.

As instruções da Tabela 3.3.13 apresentam um caso particular do acesso à variável *super* global `GLOBALS`. No caso do acesso ser a uma posição de um *array* de uma variável global, este é feito com as instruções de *array* tal como uma entrada de utilizador. O programa da Listagem 3.6 faz a atribuição de duas variáveis *super* globais a uma variável local.

O programa da Listagem 3.7 é a tradução do programa da Listagem 3.6 para HHAS, que apresenta um caso em que são usadas estas instruções.

Como uma variável global existe em todos os contextos e uma variável passada por referência existe no contexto dessa função e no contexto da função que faz a chamada, é necessário guardar o tipo que estas variáveis têm à saída de uma função. Por estas variáveis existirem para além do contexto de uma função, nesta dissertação, irão ser designadas por variáveis do exterior.

```
1 <?php
2
3     $bebida = $_POST["bebida"];
4     $GLOBALS["Menu1"]["bebida"] = htmlentities($bebida);
5     confirmaPedidoMenu1();
6     function confirmaPedidoMenu1(){
7     echo "O pedido do Menu1 com a bebida ".
          $GLOBALS["Menu1"]["bebida"] ." foi feito com sucesso";
8     }
9 ?>
```

Listagem 3.5: Exemplo de uma variável global que passa a ter o tipo `Tainted` dentro de uma função

```
1 <?php
2     $a = $GLOBALS["u"];
3     $b = $_POST["t"];
4     $l = $GLOBALS["lista"][0];
5 ?>
```

Listagem 3.6: Exemplo de programa onde são utilizadas duas variáveis *super* globais

```
1  .main <"HH\\int" "HH\\int"  > main() {
2    String "u"
3    CGetG
4    SetL $a
5    PopC
6    String "_POST"
7    BaseGC 0 Warn
8    QueryM 1 CGet ET:"t"
9    SetL $b
10   PopC
11   String "lista"
12   BaseGC 0 Warn
13   QueryM 1 CGet EI:0
14   SetL $l
15   PopC
16   Int 1
17   RetC
18 }
```

Listagem 3.7: Programa da Listagem 3.6 traduzido para HHAS

Os tipos dos parâmetros de entrada e o tipo de retorno de uma função são obtidos diretamente através do seu *type hint*. Por outro lado, os tipos à saída das variáveis do exterior, só são conhecidos depois de ser feita a sua inferência, ao longo da análise da função. No final desta análise estes tipos passam a fazer parte da nova assinatura enriquecida dessa função.

Como a assinatura de uma função pode ser alterada no final da sua análise, então, o ambiente em que está a ser feita a análise também é alterado. Isto significa que funções analisadas anteriormente não consideraram este novo ambiente que foi inferido. Para resolver este problema a abordagem utilizada passa por usar novamente um algoritmo de calculo do ponto fixo, que está definido na regra de tipos **Calc. Ponto Fixo Inf. Ambiente** na Figura 3.7. Em cada iteração da análise do programa é comparado o ambiente inicial $\Gamma$ com um ambiente inferido $\Gamma'$. Se os ambientes forem diferentes o programa é analisado novamente considerando o ambiente inferido, se os ambientes forem iguais o ponto fixo é encontrado e a análise termina. Embora não tenha sido possivel representar nas regras de sistema de tipos um parametro passado com um tipo inesperado só torna o programa vulnerável nessa iteração da análise for encontrado o ponto fixo.

Na Tabela 3.3.16, a regra de tipos da instrução `FCall`, define-se que um parâmetro com um tipo inesperado, passado na chamada a uma função, determina o programa como vulnerável. No entanto, isso só acontece se na iteração em que isso suceda, for encontrado o ponto fixo acima referido.

Na regra de tipos **Lista de Funções BT** da Figura 3.7, é possível observar que cada vez que uma função é analisada é feita uma inferência de um novo ambiente $\Gamma'$. Para garantir que o ponto fixo é encontrado na regra de tipos **Calc. Ponto Fixo Inf. Ambiente** a análise da função seguinte tem de considerar o novo ambiente $\Gamma'$.

A regra de tipos da instrução `SetM`, na Tabela 3.3.17, mantem-se igual (à da Tabela 3.3.12), caso a variável, a que é feita a atribuição, não seja uma entrada de utilizador, acrescentando apenas a premissa $\varphi.\texttt{tvariavel} \notin \texttt{TvEU}$.

Na regra de tipos da instrução `RetC`, na Tabela 3.3.15, o tipo *Função* da função *FID*, passa agora a guardar também os tipos das variáveis do exterior. Nesta regra é usada a função *éPassadaPorRef*, que verifica se uma variável é um parâmetro passado por referência nessa função.

A regra de tipos da instrução `FCall`, na Tabela 3.3.16, é agora definida tendo em conta o tipo das variáveis do exterior. Nesta regra é utilizada a função *varExtAlteraVarIDPorRef* que no mapa de variáveis do exterior (*varExt*) altera o `VarID` dos parâmetros passados por referência, pelo `VarID` da variável utilizada na chamada à função.

Tal como as variáveis locais as variáveis globais são tidas em consideração quando é feita a junção de *Frame Branches* pela função *juntaFrameBranches* apresentada no Secção 3.3.2.

Tabela 3.3.13: Funcionalidade de variáveis globais e entradas de utilizador nas instruções Get da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `CGetG` | $\Gamma \vdash P_i = (\texttt{Untainted}\ s) \cdot \beta$<br>$s \neq \epsilon\ \wedge\ \varphi = (s, \texttt{globals})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (^* TS_i[\varphi],\ \varphi) \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |
| | $\Gamma \vdash P_i = (\texttt{Untainted}\ s) \cdot \beta$<br>$s \neq \epsilon\ \wedge\ \varphi = (s, \texttt{globals})$<br>$\varphi \notin Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (\texttt{Tainted},\ \varphi) \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.14: Funcionalidade de variáveis globais e entradas de utilizador nas instruções de Mutação da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `SetG` | $tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot (\texttt{Untainted}\ s) \cdot \beta$<br>$s \neq \epsilon\ \wedge\ \varphi = (s, \texttt{globals})$<br>$\tau = ObtT(\mu)$ | $\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \tau]$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.15: Funcionalidade de variáveis globais e entradas de utilizador na instrução de Retorno da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `RetC` | $tf = \texttt{função} \ \wedge\ \gamma = \epsilon$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$ | $\forall \varphi \in Dom(TS_i)$ .<br>$(\varphi.\texttt{tvariavel} = \texttt{globals} \ \vee\ éPassadaPorRef(\varphi.\texttt{nome},\ \varkappa,\ \kappa))$ .<br>$e.varExt[\varphi \twoheadrightarrow\ ^*TS_i[\varphi]]$<br>$e = \langle varExt,\ \_,\ \_\rangle$<br>$f^{'} = \langle FID,\ \alpha,\ \varkappa,\ \kappa,\ \tau,\ e\rangle$<br>$\Gamma^{'} = \Gamma[FID \mapsto \langle C,\ f^{'}\rangle]$<br>$\dashv \Gamma^{'}$ |

Tabela 3.3.16: Funcionalidade de variáveis globais e entradas de utilizador na instrução de chamada a funções da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `FCall` ***int*** | $\Gamma \vdash pf_i = fid^{'} \cdot pf^{'}$<br>$\Gamma[fid^{'}].\texttt{tipo} = \langle \alpha^{'},\ \varkappa^{'},\ \kappa^{'},\ \gamma^{'},\ e^{'}\rangle$<br>$\Gamma \vdash P_i = \beta^{'} \bullet \beta$<br>$\boldsymbol{int} = \lvert\beta^{'}\rvert \ \wedge\ \varsigma = getListaT(\beta^{'})$<br>$\varsigma <: \alpha^{'}$ | $\Gamma \vdash P_{i+1} = \gamma^{'} \cdot \beta$<br>$\Gamma \vdash pf_{i+1} = pf^{'}$<br>$varExt^{'} = varExtAlteraVarIDPorRef(e^{'}.varExt,\ \varkappa,\ \beta^{'})$<br>$Dom(TS_{i+1}) = Dom(TS_i) \cup Dom(varExt^{'})$<br>$\forall \varphi \in Dom(varExt^{'})$ .<br>$\Gamma \vdash TS_{i+1}[\varphi] = e^{'}.varExt[\varphi]$<br>$\forall \varphi \in Dom(TS_i) \setminus Dom(varExt^{'})$ .<br>$\Gamma \vdash TS_{i+1}[\varphi] = TS_i[\varphi]$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.17: Funcionalidade de variáveis globais e entradas de utilizador nas instruções de arrays da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `BaseGC` ***int*** | $\lvert\beta^{'}\rvert = \boldsymbol{int}$<br>$\Gamma \vdash P_i = \beta^{'} \cdot (\texttt{Untainted}\ \boldsymbol{s}) \cdot \beta$<br>$\Gamma \vdash rbm_i = \epsilon$ | $\Gamma \vdash P_{i+1} = P_i$<br>$\boldsymbol{s} \notin \texttt{TvEU} \Rightarrow \Gamma \vdash rbm_{i+1} = (\boldsymbol{s},\ \texttt{globals})$<br>$\boldsymbol{s} \in \texttt{TvEU} \Rightarrow \Gamma \vdash rbm_{i+1} = (\epsilon,\ \boldsymbol{s})$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.17: Funcionalidade de variáveis globais e entradas de utilizador nas instruções de arrays da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `QueryM` ***int*** $\boldsymbol{\rho}\ \boldsymbol{\kappa}\ \boldsymbol{x}$ | $\boldsymbol{\rho} = \texttt{CGet} \ \wedge\ tf = \texttt{main} \wedge$<br>$(\kappa = \texttt{ET} \ \vee\ \kappa = \texttt{EI})$<br>$\lvert\beta^{'}\rvert = \boldsymbol{int}$<br>$\Gamma \vdash P_i = \beta^{'} \cdot \beta$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{tvariavel} \notin \texttt{TvEU} \ \wedge\ \varphi \in Dom(TS_i)$<br>$\tau = \ ^*TS_i[\varphi]$<br>$\tau \in \texttt{TArray} \ \wedge\ \tau = \texttt{UntaintedArray}$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |
| | $\boldsymbol{\rho} = \texttt{CGet}$<br>$\lvert\beta^{'}\rvert = \boldsymbol{int}$<br>$\Gamma \vdash P_i = \beta^{'} \cdot \beta$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{tvariavel} \in \texttt{TvEU}$ | $\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.17: Funcionalidade de variáveis globais e entradas de utilizador nas instruções de arrays da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| SetM ***int*** $\kappa$ $x$ | $tf = \texttt{main}$<br>$(\kappa = \texttt{ET} \ \vee \ \kappa = \texttt{EI})$<br>$\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{tvariavel} \notin \texttt{TvEU} \ \wedge \ \varphi \in Dom(TS_i)$<br>$\tau = \ ^{*}TS_i[\varphi] \wedge \tau \in$ TaintedArray<br>$\tau^{'} = ObtT(\mu) \wedge \tau^{'} = \texttt{Untainted}$ | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi \mapsto \tau[x \twoheadrightarrow \tau^{'}]]$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in \ Dom(C)$ |
| | $\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{tvariavel} \in \texttt{TvEU}$ | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$<br>$\Gamma \vdash TS_{i+1} = TS_i$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in \ Dom(C)$ |

### 3.3.6 *Aliases*

Na análise estática são tidas em conta as *aliases*. Isso significa que uma variável que é ligada a outra variável aponta para o mesmo endereço de memória de uma ou mais variáveis. Apesar de serem consideradas as instruções de *bind* e *unset*, estas instruções têm restrições para facilitar a análise estática e para garantir que não existem vulnerabilidades.

As restrições impostas passam por impedir que sejam feitos *binds* ou *unsets* dentro de uma condição ou dentro de um ciclo, inclusive em funções mutuamente recursivas, quando estas operações são feitas sobre variáveis do exterior. Isto significa que não são considerados *aliases* condicionais [1]. Para impor estas restrições nas funções mutuamente recursivas foi criado um *Call Function Graph* que não é apresentado nesta análise.

Também não são permitidas *aliases* a variáveis acedidas dinamicamente (explicado na Secção 3.3.7) ou a qualquer posição de um *array* de modo a simplificar os casos de estudo. Para além disso, não são permitidos *unsets* ou *aliases* a variáveis de entradas de utilizador de modo a promover uma boa prática de programação.

Dentro de uma função as *aliases* das variáveis do exterior não são conhecidas. Isto significa que quando é feito uma atribuição de um tipo `Tainted` a uma variável esse valor pode ser propagado para outra variável. A ferramenta lida com esta situação da seguinte forma.

Sempre que é feito uma atribuição `Tainted` a uma variável que não seja isolada do exterior, todas as variáveis que não sejam isoladas do exterior passam a ser consideradas `Tainted`. Caso a atribuição seja do tipo `Tainted TArray` essa variável passa a ter esse tipo mas todas as outras variáveis do exterior passam a ter o tipo `Tainted`. Uma variável é considerada isolada do exterior se for:

- Uma variável do exterior que já tenha sido *unset*.
- Uma variável local que não tenha sido *aliases* a uma variável não isolada do exterior.
- Uma variável local que já tenha sido *aliases* a uma variável não isolada do exterior e que posteriormente tenha sido unset.
- Uma variável do exterior que tenha sido *aliases* a uma variável isolada exterior.

O programa da Listagem 3.8 mostra duas variáveis globais que se tornaram isoladas e que por esse motivo não ficaram com o tipo malicioso quando foi feita a atribuição de um valor malicioso à variável global `idade`.

```
1  <?php
2
3  function criarUtilizador(): void{
4      unset($GLOBALS["nome"]);
5
6      $GLOBALS["nomeUtilizador"] = &$GLOBALS["nome"];
7
8      $GLOBALS["nomeUtilizador"] = htmlentities($_POST["nome"]);
9
10     $GLOBALS["idade"] = $_POST["idade"];
11
12     echo "Foi criado o utilizador " .
           $GLOBALS["nomeUtilizador"];
13 }
14 ?>
```

Listagem 3.8: Exemplo de programa em que duas variáveis globais se tornam isoladas do exterior

No programa da Listagem 3.9 podemos observar que a variável local `nIdentificacaoFiscal` deixa de ser isolada do exterior quando é *aliases* à variável global `idUtilizador`, permitindo assim que o valor malicioso recebido pela entrada de utilizador seja propagado para todas as variáveis do exterior não isoladas, neste caso a variável global `nomeUtilizador` e o seu *aliases* passam a ser maliciosos tornando o programa vulnerável ao serem passados pela função *sensitive sink* `echo`.

```
1  <?php
2
3  function criarUtilizador(): void{;
4      $GLOBALS["nomeUtilizador"] = &$GLOBALS["nome"];
5
6      $GLOBALS["nomeUtilizador"] = htmlentities($_POST["nome"]);
7
8      $nIdentificacaoFiscal = &$GLOBALS["idUtilizador"];
9      $nIdentificacaoFiscal = $_POST["nIdentificacaoFiscal"];
10
11     echo "Foi criado o utilizador " .
           $GLOBALS["nomeUtilizador"];
12 }
13 ?>
```

Listagem 3.9: Exemplo de programa em que duas variáveis globais se tornam isoladas do exterior

O programa da Listagem 3.10 mostra um caso em que é importante propagar a atribuição maliciosa para todas as variáveis que não são isoladas do exterior.

Na Tabela 3.3.18 e na Tabela 3.3.19 são apresentas as regras de tipos das instruções que permitem criar ou remover relações de *aliases*, os dois novos tipos de uma *Frame Branch* que são necessários para considerar estas instruções e três novas funções. Os dois novos tipos utilizados são o mapa de Endereços para tabela de *aliases* representada por $ta$ e o mapa $u$ de Endereços para listas, que contem tuplos compostos por um `VarID` e um tipo `T` que guarda o tipo que uma variável do exterior tinha

antes de ser *unset* ou *aliases* a outra variável. As funções utilizadas nestas tabelas são as funções *adAliases* e *rmAliases* que adicionam e removem, respectivamente, relações de *aliases* entre variáveis.

```
1  <?php
2
3    $GLOBALS["nomeUtilizador"] = &$GLOBALS["nome"];
4    criarUtilizador();
5
6  function criarUtilizador(): void{
7
8      $GLOBALS["nomeUtilizador"] = htmlentities($_POST["nome"]);
9
10     $GLOBALS["nome"] = $_POST["nomeEspecial"];
11
12   echo "Foi criado o utilizador " . $GLOBALS["nomeUtilizador"];
13 }
14 ?>
```

Listagem 3.10: Exemplo de programa que mostra a importância de propagar uma atribuição maliciosa para todas as variáveis não isoladas

No caso de uma variável local ter *aliases*, é necessário consultar a tabela de *aliases* e a lista de *unsets*, para saber se essa variável é isolada do exterior. Se todas as suas *aliases* são variáveis isoladas do exterior então a variável local é isolada do exterior. No entanto, existe um caso particular em que essa condição não é suficiente. O programa da Listagem 3.11 mostra uma variável local que é *aliases* a uma variável não isolada do exterior que posteriormente é *unset*. Neste caso não seria possível saber que a variável local não era isolada do exterior. Para contornar este problema sempre que uma variável é *aliases* a uma variável não isolada do exterior é adicionado o $\mathit{VarID}$, $\varphi = (\epsilon\ \texttt{globals})$ a essa relação de *aliases* que irá representar todas as outras variáveis do exterior que possivelmente pertencem a essa relação de *aliases*. Este procedimento pode ser observado nas regras de tipos das instruções `BindL` e `BindG` na Tabela 3.3.19.

Na Tabela 3.3.20 são apresentadas as regras de tipos das instruções de atribuição dentro de uma função, tendo agora em conta as relações de *aliases* das variáveis do exterior. Nestas instruções é utilizada a função *éIsolada* que permite verificar se uma variável é isolada do exterior e a função *setTVNIsoladas* que propaga o tipo da atribuição para todas as variáveis não isoladas, esta função faz a junção de tipos seguindo as mesmas regras da Tabela 3.3.8, fazendo a união do tipo actual de cada variável com tipo que está a ser atribuído de forma dinâmica.

O programa da Listagem 3.12 apresenta um exemplo que mostra a importância de saber o tipo de uma variável do exterior antes de ser *aliases* a outra variável. Neste caso o programa não é vulnerável porque a variável global `nome` após a chamada da função fica com o tipo `Untainted` que era o último tipo da sua variável *aliases* `nomeUtilizador`.

```
1  <?php
2
3    $GLOBALS["nomeUtilizador"] = &$GLOBALS["nome"];
4    criarUtilizador();
5
6  function criarUtilizador(): void{
7
8      $GLOBALS["nome"] = htmlentities($_POST["nome"]);
9      $nomeLocal = &$GLOBALS["nome"];
10     unset($GLOBALS["nome"]);
11     $GLOBALS["nomeUtilizador"] = $_POST["nomeEspecial"];
12
13     echo "Foi criado o utilizador " . $nomeLocal;
14 }
15 ?>
```

Listagem 3.11: Exemplo de programa que mostra a necessidade de ter um *VarID* que numa relação de *aliases* representa todas as variáveis do exterior

É importante referir que esta lista guarda estes tipos por ordem. No exemplo anterior se a relação de *aliases* fosse entre três variáveis do exterior e duas delas alteram a sua relação de *aliases* dentro da função. A variável que não alterou as suas *aliases* ficaria com o último tipo da última variável a alterar a sua relação de *aliases*.

As relações de *aliases* de variáveis do exterior e a lista $u$ da análise de uma função têm de ser utilizadas para atualizar o estado da análise após a chamada a essa função.

```
1  <?php
2
3    $GLOBALS["nomeUtilizador"] = &$GLOBALS["nome"];
4    criarUtilizador();
5
6    echo "Foi criado o utilizador " . $GLOBALS["nome"];
7
8  function criarUtilizador(): void{
9
10     $GLOBALS["nomeUtilizador"] = htmlentities($_POST["nome"]);
11
12   $GLOBALS["nomeUtilizador"] = &$GLOBALS["idUtilizador"];
13
14     $GLOBALS["nomeUtilizador"] = $_POST["nomeEspecial"];
15 }
16 ?>
```

Listagem 3.12: Exemplo de programa que mostra a importância de guarda o tipo de uma variável do exterior antes da sua relação de *aliases* ser alterada

Na Tabela 3.3.21 é apresentada novamente a regra de tipos da instrução `RetC`, onde são agora guardadas essas informações no tipo *Função* da respectiva função. Nesta instrução são utilizadas várias funções que criam sequencialmente novos tipos para atualizar e representar estas relações após a chamada da função. Nesta instru-

ção é utilizada a função *criaTAExt* para criar outra tabela de *aliases* que contem apenas as relações de *aliases* de variáveis do exterior.

A Tabela 3.3.22 apresenta a regra de tipos da instrução `FCall` tendo agora em conta as relações de *aliases*. Tal como a função *varExtAlteraVarIDPorRef*, as funções *taExtAlteraVarIDPorRef* e *uExtAlteraVarIDPorRef*, atualizam os `VarID` dos parâmetros passados por referência.

Inicialmente é utilizada a função *atualizaUnsetsTA* que remove as relações de *aliases* de uma tabela de *aliases*, de seguida é utilizada a função *atualizaUnsetsTS* que numa tabela de símbolos começa por atualizar o tipo de cada variável contida na lista de varáveis do exterior *unsets* (onde a actualização é feita pela ordem da lista) e de seguida remove todas essas variáveis da tabela de símbolo, deixando assim os seus antigos *aliases* com o tipo esperado. A função *atualizaAliasesTA* atualiza a tabela de *aliases* com as novas relações de *aliases* de variáveis do exterior criadas após a chamada à função. A função *atualizaAliasesTS* atualiza a tabela de símbolos de forma a representar as novas relações de *aliases*.

Tabela 3.3.18: Funcionalidade de *aliases* nas instruções Get da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `VGetL` $x$ | $tf = \texttt{main}$<br>$\varphi = (x,\ \texttt{globals})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (TS_i[\varphi],\ \varphi) \cdot P_i$<br>$i+1\ \in\ Dom(C)$ |
| `VGetG` $x$ | $\varphi = (x,\ \texttt{globals})$<br>$\varphi \in Dom(TS_i)$ | $\Gamma \vdash P_{i+1} = (TS_i[\varphi],\ \varphi) \cdot P_i$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.19: Funcionalidade e instruções de *aliases* da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `BindL` $x$ | $tf = \texttt{main}$<br>$\varphi' = (x,\ \texttt{globals})$<br>$\Gamma \vdash P_i = (\psi\ \varphi) \cdot \beta$ | $\Gamma \vdash P_{i+1} = TS_i[\varphi] \cdot P_i$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi' \mapsto \psi]$<br>$\Gamma \vdash ta_{i+1} = adAliases(ta_i,\ \varphi,\ \varphi')$<br>$i+1\ \in\ Dom(C)$ |
| | $tf = \texttt{função}$<br>$\varphi' = (x,\ \texttt{local})$<br>$\neg éPassadaPorRef(x,\ \varkappa,\ \kappa)$<br>$\Gamma \vdash P_i = (\psi\ \varphi) \cdot \beta$ | $\Gamma \vdash P_{i+1} = TS_i[\varphi] \cdot P_i$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi' \mapsto \psi]$<br>$(\neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)) \Rightarrow \begin{pmatrix} \varphi'' = (\epsilon\ \texttt{globals}) \\ ta'_{i+1} = adAliases(ta_i,\ \varphi,\ \varphi'') \end{pmatrix}$<br>$(éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)) \Rightarrow ta'_{i+1} = ta_i$<br>$\Gamma \vdash ta_{i+1} = adAliases(ta'_{i+1},\ \varphi,\ \varphi')$<br>$i+1\ \in\ Dom(C)$ |
| | $tf = \texttt{função}$<br>$\varphi' = (x,\ \texttt{local})$<br>$éPassadaPorRef(x,\ \varkappa,\ \kappa)$<br>$\Gamma \vdash P_i = (\psi\ \varphi) \cdot \beta$ | $\Gamma \vdash P_{i+1} = TS_i[\varphi] \cdot P_i$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi' \mapsto \psi]$<br>$(\neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)) \Rightarrow \begin{pmatrix} \varphi'' = (\epsilon\ \texttt{globals}) \\ ta'_{i+1} = adAliases(ta_i,\ \varphi,\ \varphi'') \end{pmatrix}$<br>$(éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)) \Rightarrow ta'_{i+1} = ta_i$<br>$\Gamma \vdash ta_{i+1} = adAliases(ta'_{i+1},\ \varphi,\ \varphi')$<br>$(\varphi' \in Dom(TS_i)\ \wedge \tau = {}^*TS_i[\varphi']\ \wedge\ (\varphi',\ \_) \notin u_i)$<br>$\Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi',\ \tau)$<br>$(\varphi' \notin Dom(TS_i)\ \wedge\ (\varphi',\ \_) \notin u_i) \Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi',\ \texttt{Tainted})$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.19: Funcionalidade e instruções de *aliases* da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `BindG` | $tf = \texttt{função}$<br>$\varphi' = (s,\ \texttt{globals})$<br>$\Gamma \vdash P_i = (\psi\ \varphi) \cdot (\texttt{Untainted}\ s) \cdot \beta$ | $\Gamma \vdash P_{i+1} = (\psi\ \varphi) \cdot P_i$<br>$\Gamma \vdash TS_{i+1} = TS_i[\varphi' \mapsto \psi]$<br>$(\neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i))$<br>$\Rightarrow \begin{pmatrix} \varphi'' = (\epsilon\ \texttt{globals}) \\ ta'_{i+1} = adAliases(ta_i,\ \varphi,\ \varphi'') \end{pmatrix}$<br>$(éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i))$<br>$\Rightarrow ta'_{i+1} = ta_i$<br>$\Gamma \vdash ta_{i+1} = adAliases(ta'_{i+1},\ \varphi,\ \varphi')$<br>$(\varphi' \in Dom(TS_i)\ \wedge \tau =^* TS_i[\varphi']\ \wedge\ (\varphi',\ \_) \notin u_i)$<br>$\Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi',\ \tau)$<br>$(\varphi' \notin Dom(TS_i)\ \wedge\ (\varphi',\ \_) \notin u_i)$<br>$\Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi',\ \texttt{Tainted})$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.19: Funcionalidade e instruções de *aliases* da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `UnsetL` $\boldsymbol{x}$ | $tf = \texttt{função}$<br>$\varphi = (\boldsymbol{x},\ \texttt{local})$<br>$éPassadaPorRef(\boldsymbol{x},\ \varkappa,\ \kappa)$ | $\Gamma \vdash ta_{i+1} = rmAliases(ta_i,\ \varphi)$<br>$(\varphi \in Dom(TS_i))$<br>$\Rightarrow \begin{pmatrix} \tau = {}^*TS_i[\varphi] \\ \Gamma \vdash TS_{i+1} = TS_i \setminus \varphi \end{pmatrix}$<br>$(\varphi \notin Dom(TS_i)) \Rightarrow \tau = \texttt{Tainted}$<br>$((\varphi, \_) \notin u_i) \Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi', \tau)$<br>$i+1 \in Dom(C)$ |
| `UnsetG` | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = (\texttt{Untainted}\ \boldsymbol{s}) \cdot \beta$<br>$\varphi = (\boldsymbol{s},\ \texttt{globals})$ | $\Gamma \vdash ta_{i+1} = rmAliases(ta_i,\ \varphi)$<br>$(\varphi \in Dom(TS_i))$<br>$\Rightarrow \begin{pmatrix} \tau =^* TS_i[\varphi] \\ \Gamma \vdash TS_{i+1} = TS_i \setminus \varphi \end{pmatrix}$<br>$(\varphi \notin Dom(TS_i)) \Rightarrow \tau = \texttt{Tainted}$<br>$((\varphi, \_) \notin u_i) \Rightarrow \Gamma \vdash u_{i+1} = u_i \cdot (\varphi, \tau)$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.20: Funcionalidade de *aliases* nas instruções de Mutação da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `SetL` $\boldsymbol{x}$ | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\varphi = (\boldsymbol{x},\ \texttt{local})$<br>$éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$ | $\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \tau]$<br>$\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |
| | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\varphi = (\boldsymbol{x},\ \texttt{local})$<br>$\neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$ | $\Gamma \vdash TS_{i+1} = setTVNIsoladas(\tau,\ TS_i,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$<br>$\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.20: Funcionalidade de *aliases* nas instruções de Mutação da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `SetG` | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = \mu \cdot (\texttt{Untainted}\ \boldsymbol{s}) \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\varphi = (\boldsymbol{s},\ \texttt{globals})$<br>$éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$ | $\Gamma \vdash TS_{i+1} = TS_i[\varphi \twoheadrightarrow \tau]$<br>$\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |
| | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = \mu \cdot (\texttt{Untainted}\ \boldsymbol{s}) \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\varphi = (\boldsymbol{s},\ \texttt{globals})$<br>$\neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$ | $\Gamma \vdash TS_{i+1} = setTVNIsoladas(\tau,\ TS_i,\ \varkappa,\ \kappa,\ ta_i,\ u_i)$<br>$\Gamma \vdash P_{i+1} = \tau \cdot \beta$<br>$i+1 \in Dom(C)$ |

Tabela 3.3.21: Funcionalidade de *aliases* na instrução de Retorno da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `RetC` | $tf = \texttt{função} \ \wedge\ \gamma = \epsilon$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$ObtT(\mu) = \tau$ | $taExt = criaTAExt(ta_i,\ \varkappa,\ \kappa)$<br>$\forall \varphi \in Dom(TS_i)\ .$<br>$(\varphi.\texttt{tvariavel} = \texttt{globals} \ \vee\ éPassadaPorRef(\varphi.\texttt{nome},\ \varkappa,\ \kappa))\ .$<br>$e.varExt[\varphi \twoheadrightarrow\ ^* TS_i[\varphi]]$<br>$e = \langle varExt,\ taExt,\ ui\rangle$<br>$f^{'} = \langle \alpha,\ \varkappa,\ \kappa,\ \tau,\ e\rangle$<br>$\Gamma^{'} = \Gamma[FID \mapsto \langle C,\ f^{'}\rangle]$<br>$\dashv \Gamma^{'}$ |

Tabela 3.3.22: Funcionalidade de *aliases* na instrução de chamada a funções da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | **restrições na linha i** | **restrições no sucessor de i** |
|---|---|---|
| `FCall` ***int*** | $tf = \texttt{função}$<br>$\Gamma \vdash pf_i = fid^{'} \cdot pf^{'}$<br>$\Gamma[fid^{'}].\texttt{tipo} = \langle \alpha^{'},\ \varkappa^{'},\ \kappa^{'},\ \gamma^{'},\ e^{'}\rangle$<br>$\Gamma \vdash P_i = \beta^{'} \bullet \beta$<br>$\textbf{\textit{int}} = \lvert\beta^{'}\rvert \ \wedge\ \varsigma = getListaT(\beta^{'})$<br>$\varsigma <: \alpha^{'}$ | $\Gamma \vdash P_{i+1} = \gamma^{'} \cdot \beta$<br>$\Gamma \vdash pf_{i+1} = pf^{'}$<br>$varExt^{'} = varExtAlteraVarIDPorRef(e^{'}.varExt,\ \varkappa,\ \beta^{'})$<br>$taExt^{'} = taExtAlteraVarIDPorRef(e^{'}.taExt,\ \varkappa^{'},\ \beta^{'})$<br>$uExt^{'} = uExtAlteraVarIDPorRef(e^{'}.uExt,\ \varkappa^{'},\ \beta^{'})$<br>$ta^{'}_{i+1} = atualizaUnsetsTA(uExt^{'},\ ta_i)$<br>$TS^{'}_{i+1} = atualizaUnsetsTS(uExt^{'},\ TS_i)$<br>$\Gamma \vdash ta_{i+1} = atualizaAliasesTA(taExt^{'},\ ta^{'}_{i+1})$<br>$TS^{''}_{i+1} = atualizaAliasesTS(taExt^{'},\ TS^{'}_{i+1})$<br>$ta_i = atualizaTA(taExt^{'},\ ta_i\ TSi)$<br>$Dom(TS_{i+1}) = Dom(TS_i) \cup Dom(varExt^{'})$<br>$\forall \varphi \in Dom(varExt^{'})\ .$<br>$\Gamma \vdash TS_{i+1}[\varphi] = TS^{''}_{i+1}[\varphi \twoheadrightarrow varExt^{'}[\varphi]]$<br>$\forall \varphi \in Dom(TS^{''}_{i+1}) \setminus Dom(varExt^{'})\ .$<br>$\Gamma \vdash TS_{i+1}[\varphi] = TS^{''}_{i+1}[\varphi]$<br>$\Gamma \vdash u_{i+1} = u_i + (uExt^{'} - u_i)$<br>$i + 1 \ \in\ Dom(C)$ |

### 3.3.7 Variáveis acedidas dinamicamente

Variáveis acedidas dinamicamente são outra funcionalidade do PHP que o torna tão dinâmico. Esta funcionalidade permite que uma operação de *get* ou *set* feita a uma variável que não é explicitamente declarada no código, mas sim definida em tempo de execução. Uma variável acedida dinamicamente pode ser uma *variable variable*[1](variável com nome variável), onde o nome de uma variável é variável por ser atribuído dinamicamente, ou uma posição de um *array* acedida dinamicamente. A ferramenta consegue analisar programas que utilizam este tipo de variáveis. Para isso, se for feita uma atribuição do tipo `Tainted` a uma variável global acedida dinamicamente, a análise prossegue da seguinte forma nos seguintes casos:

1. No caso de ser uma variável com nome variável, todas as variáveis globais e variáveis não isoladas do exterior passam a ter o tipo `Tainted`.

2. No caso da variável ser um elemento desconhecido de um *array* conhecido. Todas as posições desse *array* passam a ser do tipo `Tainted` e o seu tipo de *array* passa a ser `Tainted`.

3. No caso da variável ser um elemento conhecido de um *array* desconhecido. Todas as variáveis globais e variáveis não isoladas que são *arrays* passam a ter essa posição `Tainted` e o seu tipo de *array* passa a ser `Tainted`. As variáveis que não são *array* passam a ter o tipo `Tainted`.

4. No caso da variável ser um elemento desconhecido de um *array* desconhecido. Todas as variáveis globais e variáveis não isoladas passam a ter o tipo `Tainted`.

As Listagens seguintes mostram casos em que um programa utiliza variáveis acedidas dinamicamente.

No programa da Listagem 3.13 são feitas três atribuições a variáveis, onde a última atribuição é dinâmica. Inicialmente a variável `$cidade_2` tem o tipo `Untainted`, mas após ser feita uma atribuição à variável com nome variável `$nome_da_variavel` de um valor malicioso, a variável `$cidade_2` passa a ter o tipo `Tainted` por não se saber a que variável foi feita a atribuição. Este programa é vulnerável a um ataque de XSS na linha 10.

[1] `https://www.php.net/manual/en/language.variables.variable.php`

```
1  <?php
2
3      $cidade_2 = "Lisboa";
4
5      echo "A cidade 2 é". $cidade_2;
6
7      $nome_da_variavel = "cidade_2";
8      $$nome_da_variavel = $_POST["cidade"];
9
10     echo "A cidade 2 é". $cidade_2;
11 ?>
```

Listagem 3.13: Exemplo de programa onde é feita uma atribuição maliciosa a uma variável com nome variável

No programa da Listagem 3.14 é feita uma passagem sobre um *array* do tipo `Untainted`. Apesar de não se saber a que posição do *array* está a ser feita a operação `QueryM CGet`, na Listagem 3.15 na linha 26, sabe-se que todas as posições do *array* têm o tipo `Untainted` e, portanto, o tipo adicionado ao topo da pilha é `Untainted`.

No programa da Listagem 3.16 é feito o acesso de forma dinâmica a uma posição de um *array* do tipo `Tainted`, o que significa que o tipo obtido é do tipo `Tainted` apesar do *array* `$utilizadores` ter quase todas as posições com tipo `Untainted`. A análise detecta assim que este programa é vulnerável a um ataque de XSS na linha 10.

Da Tabela 3.3.23 à 3.3.25 são apresentadas as regras de tipos das instruções que permitem aceder a variáveis de forma dinâmica. Nestas regras é utilizado o novo tipo booleano *adt* contido numa *Frame Branch*. Na análise de uma função este booleano começa por ser 0 como é possível observar na regra de tipos **Código Função** na Figura 3.7. Este booleano passa a ser verdadeiro se for feita alguma atribuição maliciosa de forma dinâmica a uma variável global.

A Tabela 3.3.27 apresenta a regra de tipos da instrução `FCall` tendo agora em conta o tipo *adt*. Caso este booleano seja verdadeiro então todas as variáveis globais passam a ter o tipo `Tainted`.

```
1  <?php
2
3      $utilizadores = array();
4      $utilizadores[0] = "João";
5      $utilizadores[1] = "José";
6
7      for ($i = 0; $i < count($utilizadores); $i++) {
8          echo "| " . $utilizadores[$i];
9      }
10 ?>
```

Listagem 3.14: Programa onde é utilizada uma posição desconhecida de um *array* do tipo `Untainted`

```
1  .adata A_0 = """a:0:{}""";
2  .main <"HH\\int" "HH\\int"  > main() {
3    Array @A_0
4    SetL $utilizadores
5    PopC
6    String "João"
7    BaseL $utilizadores Define
8    SetM 0 EI:0
9    PopC
10   String "José"
11   BaseL $utilizadores Define
12   SetM 0 EI:1
13   PopC
14   Int 0
15   SetL $i
16   PopC
17   FPushFuncD 1 "count"
18   CGetL $utilizadores
19   FCall <> 1 1 - "" ""
20   CGetL2 $i
21   Lt
22   JmpZ L0
23 L1:
24   String "| "
25   BaseL $utilizadores Warn
26   QueryM 0 CGet EL:$i
27   Concat
28   Print
29   PopC
30   IncDecL $i PostInc0
31   PopC
32   FPushFuncD 1 "count"
33   CGetL $utilizadores
34   FCall <> 1 1 - "" ""
35   CGetL2 $i
36   Lt
37   JmpNZ L1
38 L0:
39   Int 1
40   RetC
41 }
```

Listagem 3.15: Programa da Listagem 3.14 traduzido para HHAS

```
1  <?php
2
3      $utilizadores = array();
4      $utilizadores[0] = "João";
5      $utilizadores[1] = "José";
6      $utilizadores[2] = "Moreira";
7      $utilizadores[3] = $_POST["u"];
8
9      for ($i = 0; $i < count($utilizadores); $i++) {
10         echo "| " . $utilizadores[$i];
11     }
12 ?>
```

Listagem 3.16: Programa onde é utilizada uma posição desconhecida de um *array* do tipo `Tainted`

Tabela 3.3.23: Funcionalidade de variáveis acedidas dinamicamente nas instruções Get da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `CGetN` | $\Gamma \vdash P_i = (\texttt{Untainted}\ \epsilon) \cdot \beta$ | $\varphi = (\epsilon,\ \texttt{local})$<br>$\Gamma \vdash P_{i+1} = (\texttt{Tainted}\ \varphi) \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |
| `CGetG` | $\Gamma \vdash P_i = (\texttt{Untainted}\ \epsilon) \cdot \beta$ | $\varphi = (\epsilon,\ \texttt{globals})$<br>$\Gamma \vdash P_{i+1} = (\texttt{Tainted}\ \varphi) \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.24: Funcionalidade de variáveis acedidas dinamicamente nas instruções de Mutação da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `SetN` | $\Gamma \vdash P_i = \mu \cdot \mu' \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau = \texttt{Untainted} \vee \tau = \texttt{UntaintedArray}$ | $\Gamma \vdash P_{i+1} = \mu \cdot \beta$<br>$i+1\ \in\ Dom(C)$ |
| | $tf = \texttt{main}$<br>$\Gamma \vdash P_i = \mu \cdot \mu' \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau = \texttt{Tainted} \vee \tau \in$ TaintedArray | $\forall \varphi \in Dom(TS_i)$<br>$\Rightarrow \Gamma \vdash TS_{i+1}[\varphi] = TS_i[\varphi \twoheadrightarrow \texttt{Tainted}]$<br>$\Gamma \vdash P_{i+1} = \mu \cdot \beta$<br>$\forall j \in Dom(u_i)$<br>$\Rightarrow \begin{pmatrix}(\varphi,\ \tau) = u_i[j] \\ \Gamma \vdash u_{i+1}[j] = (\varphi,\ \texttt{Tainted})\end{pmatrix}$<br>$i+1\ \in\ Dom(C)$ |
| `SetG` | $tf = \texttt{função}$<br>$\Gamma \vdash P_i = \mu \cdot (\texttt{Untainted}\ \epsilon) \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$\tau = \texttt{Tainted} \vee \tau \in$ TaintedArray | $\forall \varphi \in Dom(TS_i)\ .$<br>$\begin{pmatrix}\varphi.\texttt{tvariavel} = \texttt{globals} \vee \\ \neg éIsolada(\varphi,\ \varkappa,\ \kappa,\ ta_i,\ u_i)\end{pmatrix}$<br>$\Rightarrow \Gamma \vdash TS_{i+1}[\varphi] = TS_i[\varphi \twoheadrightarrow \texttt{Tainted}]$<br>$\Gamma \vdash P_{i+1} = \mu \cdot \beta$<br>$\Gamma \vdash adt_{i+1} = 1$<br>$\forall j \in Dom(u_i)$<br>$\Rightarrow \begin{pmatrix}(\varphi,\ \tau) = u_i[j] \\ \Gamma \vdash u_{i+1}[j] = (\varphi,\ \texttt{Tainted})\end{pmatrix}$<br>$i+1\ \in\ Dom(C)$ |

Tabela 3.3.25: Funcionalidade de variáveis acedidas dinamicamente nas instruções de *arrays* da SE

$\Gamma,\ \langle i,\ TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | sub-restrições na linha i | restrições no sucessor de i |
|---|---|---|---|
| QueryM ***int*** $\boldsymbol{\rho}$ $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $\boldsymbol{\rho} = \texttt{CGet}$<br>$\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \beta^{'} \cdot \beta$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{tvariavel} \notin \texttt{TvEU}$ | $tf = \texttt{main}$<br>$(\varphi.\texttt{nome} = \epsilon \ \vee\ \varphi \notin Dom(TS_i))$ | $\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |
| | | $tf = \texttt{main}$<br>$\kappa = \texttt{EL}$<br>$\varphi.\texttt{nome} \neq \epsilon \ \wedge\ \varphi \notin Dom(TS_i)$<br>$\tau = \ ^*TS_i[\varphi]$<br>$\tau = \texttt{UntaintedArray}$ | $\Gamma \vdash P_{i+1} = \texttt{Untainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |
| | | $tf = \texttt{main}$<br>$\kappa = \texttt{EL}$<br>$\varphi.\texttt{nome} \neq \epsilon \ \wedge\ \varphi \notin Dom(TS_i)$<br>$\tau = \ ^*TS_i[\varphi]$<br>$\tau \in$ TaintedArray | $\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.25: Funcionalidade de variáveis acedidas dinamicamente nas instruções de *arrays* da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle \{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| SetM ***int*** $\boldsymbol{\kappa}$ $\boldsymbol{x}$ | $tf = \texttt{função}$<br>$\boldsymbol{\rho} = \texttt{CGet}$<br>$\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$(\tau = \texttt{Tainted} \ \vee\ \tau \in$ TaintedArray$)$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{nome} = \epsilon \ \wedge\ \varphi.\texttt{tvariavel} = \texttt{local}$ | $condição = \begin{pmatrix} \exists \varphi^{'} \in Dom(TS_i). \\ (\neg éIsolada(\varphi^{'},\ \varkappa,\ \kappa,\ ta_i,\ ui)) \end{pmatrix}$<br>$condição \ \wedge \begin{pmatrix} \forall \varphi^{'} \in Dom(TS_i). \\ \begin{pmatrix} \varphi.\texttt{tvariavel} = \texttt{local} \ \vee \\ \neg éIsolada(\varphi^{'},\ \varkappa,\ \kappa,\ ta_i,\ ui) \end{pmatrix} \\ \Rightarrow \Gamma \vdash TS_{i+1}[\varphi^{'}] = TS_i[\varphi^{'} \twoheadrightarrow \texttt{Tainted}] \\ \\ \forall j \in Dom(u_i). \\ \begin{pmatrix} (\varphi^{'},\ \tau) = u_i[j] \\ \varphi^{'}.\texttt{tvariavel} = \texttt{local} \\ \Rightarrow \Gamma \vdash u_{i+1}[j] = (\varphi^{'},\ \texttt{Tainted}) \end{pmatrix} \end{pmatrix}$<br>$\neg condição \wedge \forall \varphi^{'} \in Dom(TS_i).$<br>$(\varphi^{'}.\texttt{tvariavel} = \texttt{local}) \Rightarrow \Gamma \vdash TS_{i+1}[\varphi^{'}] = TS_i[\varphi^{'} \twoheadrightarrow \texttt{Tainted}]$<br>$\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.25: Funcionalidade de variáveis acedidas dinamicamente nas instruções de *arrays* da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `SetM` ***int*** $\kappa$ ***x*** | $tf = \texttt{função}$<br>$\boldsymbol{\rho} = \texttt{CGet}$<br>$\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$(\tau = \texttt{Tainted} \ \vee\ \tau \in \text{TaintedArray})$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{nome} = \epsilon \ \wedge\ \varphi.\texttt{tvariavel} = \texttt{globals}$ | $\forall \varphi^{'} \in Dom(TS_i)$<br>$\Rightarrow \Gamma \vdash TS_{i+1}[\varphi^{'}] = TS_i[\varphi^{'} \twoheadrightarrow \texttt{Tainted}]$<br>$\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$\Gamma \vdash adt_{i+1} = 1$<br>$\forall j \in Dom(u_i)$<br>$\Rightarrow \begin{pmatrix} (\varphi^{'},\ \tau) = u_i[j] \\ \Gamma \vdash u_{i+1}[j] = (\varphi^{'},\ \texttt{Tainted}) \end{pmatrix}$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.25: Funcionalidade de variáveis acedidas dinamicamente nas instruções de *arrays* da SE (cont.)

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `SetM` ***int*** $\kappa$ ***x*** | $tf = \texttt{main}$<br>$\boldsymbol{\rho} = \texttt{CGet} \ \wedge\ \kappa = \texttt{EL}$<br>$\lvert\beta^{'}\rvert = \textbf{\textit{int}}$<br>$\Gamma \vdash P_i = \mu \cdot \beta^{'} \cdot \beta$<br>$\tau = ObtT(\mu)$<br>$(\tau = \texttt{Tainted} \ \vee\ \tau \in \text{TaintedArray})$<br>$\Gamma \vdash rbm = \varphi$<br>$\varphi.\texttt{nome} \neq \epsilon \ \wedge\ \varphi.\texttt{tvariavel} \notin \texttt{TvEU}$ | $\forall \varphi^{'} \in Dom(TS_i)$<br>$\Rightarrow \Gamma \vdash TS_{i+1} = TS_i[\varphi^{'} \twoheadrightarrow (\texttt{TaintedArray}\ \epsilon)]$<br>$\Gamma \vdash P_{i+1} = \texttt{Tainted} \cdot \beta$<br>$\Gamma \vdash rbm_{i+1} = \epsilon$<br>$i+1 \ \in\ Dom(C)$ |

Tabela 3.3.26: Funcionalidade de variáveis acedidas dinamicamente na instrução de Retorno da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `RetC` | $tf = \texttt{função} \ \wedge\ \gamma = \epsilon$<br>$\Gamma \vdash P_i = \mu \cdot \beta$<br>$\tau = ObtT(\mu)$ | $taExt = criaTAExt(ta_i,\ \varkappa,\ \kappa)$<br>$\forall \varphi \in Dom(TS_i)\ .$<br>$(\varphi.\texttt{tvariavel} = \texttt{globals} \ \vee\ éPassadaPorRef(\varphi.\texttt{nome},\ \varkappa,\ \kappa))\ .$<br>$e.varExt[\varphi \twoheadrightarrow\ ^{*}TS_i[\varphi]]$<br>$e = \langle varExt,\ taExt,\ u_i,\ adt_i\rangle$<br>$f^{'} = \langle \alpha,\ \varkappa,\ \kappa,\ \tau,\ e\rangle$<br>$\Gamma^{'} = \Gamma[FID \mapsto \langle C,\ f^{'}\rangle]$<br>$\dashv \Gamma^{'}$ |

Tabela 3.3.27: Funcionalidade de variáveis acedidas dinamicamente na instrução de chamada a funções da SE

$\Gamma,\ \langle i, TS,\ P,\ rbm,\ pf,\ ta,\ u,\ adt\rangle_b,\ B \vdash C : \langle\{|tf, nf|\}_{FID},\ \alpha,\ \varkappa,\ \kappa,\ \gamma,\ e\rangle$

| $C[i]$ | restrições na linha i | restrições no sucessor de i |
|---|---|---|
| `FCall` ***int*** | $tf = \texttt{função}$<br>$\Gamma \vdash pf_i = fid^{'} \cdot pf^{'}$<br>$\Gamma[fid^{'}].\texttt{tipo} = \langle \alpha^{'},\ \varkappa^{'},\ \kappa^{'},\ \gamma^{'},\ e^{'}\rangle$<br>$\Gamma \vdash P_i = \beta^{'} \bullet \beta$<br>$\textbf{\textit{int}} = \lvert\beta^{'}\rvert\ \wedge\ getListaT(\beta^{'}) = \varsigma$<br>$\varsigma <: \alpha^{'}$ | $\Gamma \vdash P_{i+1} = \gamma^{'} \cdot \beta$<br>$\Gamma \vdash pf_{i+1} = pf^{'}$<br>$e^{'}.adtExt\ \wedge\ \forall \varphi \in Dom(TS_i)\ .$<br>$\left(\begin{array}{l} condição = \left(\begin{array}{l}\varphi.\texttt{tvariavel} = \texttt{globals} \vee \\ \neg éIsolada(\varphi,\ \varkappa^{'},\ \kappa^{'},\ ta_i,\ u_i)\end{array}\right) \\ condição \Rightarrow \Gamma \vdash TS^{'}_{i+1} = TS_i[\varphi \twoheadrightarrow \texttt{Tainted}] \\ \neg condição \Rightarrow \Gamma \vdash TS^{'}_{i+1} = TS_i[\varphi] \end{array}\right)$<br>$e^{'}.adtExt\ \Rightarrow \Gamma \vdash e_{i+1} = 1$<br>$\neg e^{'}.adtExt\ \Rightarrow \Gamma \vdash TS^{''}_{i+1} = TS_i$<br>$varExt^{'} = varExtAlteraVarIDPorRef(e^{'}.varExt,\ \varkappa^{'},\ \beta^{'})$<br>$taExt^{'} = taExtAlteraVarIDPorRef(e^{'}.taExt,\ \varkappa^{'},\ \beta^{'})$<br>$uExt^{'} = uExtAlteraVarIDPorRef(e^{'}.uExt,\ \varkappa^{'},\ \beta^{'})$<br>$ta^{'}_{i+1} = atualizaUnsetsTA(uExt^{'},\ ta_i)$<br>$TS^{''}_{i+1} = atualizaUnsetsTS(uExt^{'},\ TS_i)$<br>$\Gamma \vdash ta_{i+1} = atualizaAliasesTA(taExt^{'},\ ta^{'}_{i+1})$<br>$TS^{''}_{i+1} = atualizaAliasesTS(taExt^{'},\ TS^{''}_{i+1})$<br>$ta_i = atualizaTA(taExt^{'},\ ta_i\ TSi)$<br>$Dom(TS_{i+1}) = Dom(TS_i) \cup Dom(varExt^{'})$<br>$\forall \varphi \in Dom(varExt^{'})\ .$<br>$\quad \Gamma \vdash TS_{i+1}[\varphi] = TS^{'''}_{i+1}[\varphi \twoheadrightarrow varExt^{'}[\varphi]]$<br>$\forall \varphi \in Dom(TS^{'''}_{i+1}) \setminus Dom(varExt^{'})\ .$<br>$\quad \Gamma \vdash TS_{i+1}[\varphi] = TS^{'''}_{i+1}[\varphi]$<br>$\Gamma \vdash u_{i+1} = u_i + (uExt^{'} - u_i)$<br>$i + 1\ \in\ Dom(C)$ |

# Capítulo 4

# Desenho e Implementação do Detetor de Vulnerabilidades HipHop

Este capítulo descreve a solução que propomos para deteção de vulnerabilidades de SQLI e XSS em programas PHP compilados para a linguagem de baixo nível HHAS, baseada em verificação de tipos de dados através do sistema de tipos apresentado no Capítulo 3, e a ferramenta DVHipHop(Detector de Vulnerabilidades HipHop) que a implementa. Na Secção 4.1 é apresentado o objetivo e o funcionamento generalizado da solução proposta. Na Secção 4.2 é ilustrado um exemplo de como é efectuada a deteção de vulnerabilidades pela solução proposta. Na Secção 4.3, é explicada a implementação do DVHipHop, onde são representadas e explicadas as principais estruturas de dados utilizadas pelo DVHipHop.

## 4.1 Detetor de Vulnerabilidades HipHop

Nesta secção irá ser apresentado o objetivo e o funcionamento generalizado da solução proposta.

O objetivo principal da solução proposta é realizar estaticamente a inferência de tipos de um programa e conhecer quais são as variáveis com tipos maliciosos para garantir que essas varáveis não são utilizadas num sítio do programa onde seja esperada uma variável com um tipo não malicioso.

A arquitetura da solução que propomos tem apenas acesso ao código fonte e a um ficheiro de configuração com uma lista das interfaces de chamada de funções. O código fonte irá ser traduzido para uma linguagem de baixo nível que é depois organizada em estruturas de dados, como por exemplo numa AST, que facilitem a sua interpretação e análise. Após toda a informação estaticamente disponível estar organizada, é utilizado um analisador estático para fazer uma análise de comprometimento (*taint analysis*) baseada numa análise de tipos. Durante esta análise é

utilizado o ficheiro de configuração.

A Figura 4.1 ilustra a arquitetura da solução proposta. O DVHipHop é constituído por dois componentes principais, o *Gerador de Código Intermédio* e o *Analisador Estático de Código*, que permite converter o código PHP/Hack para a linguagem intermédia HHAS (linguagem que descrevemos na Secção 3.1) e identificar vulnerabilidades pela análise de tipos e de comprometimento, respectivamente. De seguida apresentamos os módulos que os compõem e as seis estruturas de dados que usam.

Os três módulos pertencentes aos dois componentes são utilizados pela seguinte ordem:

1. **Tradução para HHAS** - Trata da tradução e otimização do código fonte PHP e Hack para a linguagem de baixo nível HHAS. No final é produzido um ficheiro escrito em HHAS que a partir do qual será obtida a AST para a realização da análise semântica (3º módulo).

2. **Criação da AST** - Através da ferramenta Antlr [24] são criados um *Lexer* e um *Parser* para HHAS e que por sua vez permitem criar uma AST, a qual permitirá a realização da análise de tipos. A AST é criada pela passagem sobre a árvore de *parser* que o *parser* do Antlr gerou em memória.

3. **Análise de Tipos e Análise de Comprometimento** - Este módulo utiliza uma análise estática de comprometimento baseada em análise de tipos (explicado no Capítulo 3), para verificar se não existe nenhuma variável com o tipo `Tainted` que seja usada numa função sensível (*sensitive sink*) sítio onde é esperado um tipo `Untainted` ou *SString*. Para uma análise mais precisa é feita uma passagem inicial na AST que tem como objetivo obter informação sobre a estrutura do programa em análise e criar as estruturas de dados necessárias para a realização da análise. Os ramos da AST que equivalem a uma função são organizados nestas estruturas de dados. Isto permite que na análise sejam feitas passagens nesses ramos analisando cada função de forma individual. Estas passagens na AST são apresentadas de seguida, sendo que a análise propriamente dita é apresentada em mais detalhe na Secção 4.2

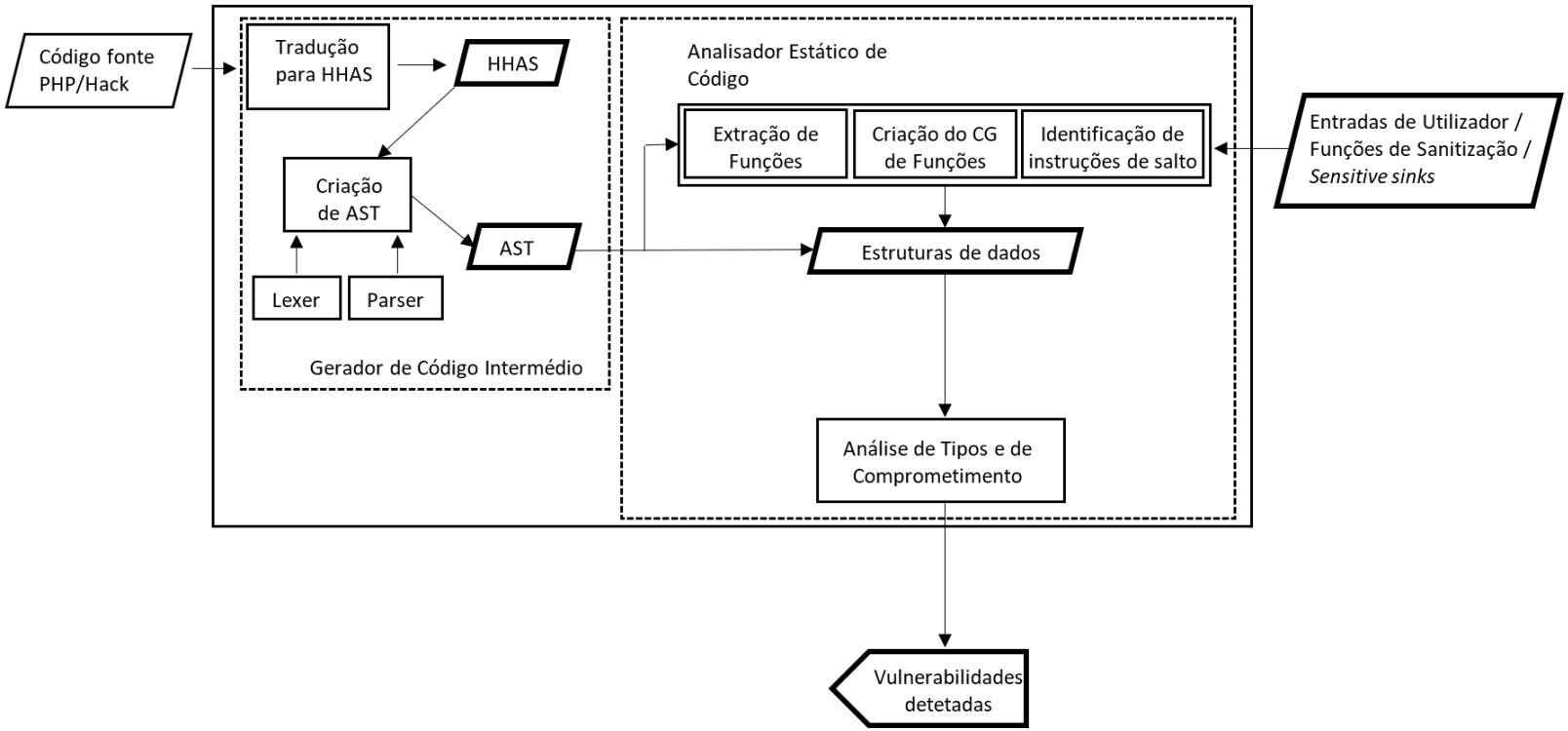


Figura 4.1: Arquitetura da solução DVHipHop

*Extração das Funções*: A primeira passagem pela AST permite conhecer todas as funções que vão ser analisadas, onde é recolhido o *type hint* dos seus parâmetros e do seu retorno tal como o ramo da AST equivalente a essa função. Para além disto, esta passagem tem também o papel de criar um *Call Function Graph* (CG) que permite garantir que não são criadas *may-aliases* de variáveis do exterior em funções mutuamente recursivas. Por fim, nesta passagem também é realizado o mapeamento das instruções de salto que há no programa para o respetivo nó na AST.
*Análise de Funções*: A análise de comprometimento é feita fazendo múltiplas passagens na AST correspondente a cada função. Estas passagens analisam cada função individualmente por ordem alfabética de forma a tornar a análise determinística. Quando é efectuada uma passagem para analisar uma função podem ser feitas inferências que enriquecem a sua assinatura. Por este motivo, se for feita uma chamada a uma função que ainda não tenha sido analisada é considerada a assinatura recolhida na primeira passagem (*Extração de Funções*). Se na análise do programa, a assinatura de uma ou mais funções for alterada então o programa é analisado novamente considerando as novas assinaturas.

Para realizar a análise de tipos e comprometimento as principais estruturas de dados utilizadas e geradas são:

- *Mapa com a interface de chamada das funções* que já foram analisadas, tal como para todas as funções *built-in* do PHP e funções de bibliotecas externas, definidas no ficheiro, onde estão incluídas as funções de sanitização e *sensitive sinks*.
- *Mapa de etiquetas* para o número da instrução dessa etiqueta (etiquetas que definem para onde são feitas as instruções de salto).
- *Frame de estado* que irá conter toda a informação do estado da análise de uma determinada função, para um fluxo do programa.
- *Mapa de etiquetas para Frame de estado* que contêm informações do estado da análise para diferentes fluxos do programa.
- *Pilha de avaliação* que contem objetos *HHTAType*. Esta pilha vai sendo atualizada consoante as operações HHAS vão sendo analisadas.
- *Tabela de símbolos* que consiste num mapa de *VariableID* (identificador de uma variável) para *VariantBox*. Esta tabela permite guardar os tipos de cada variável.
- *Tabela de aliases* para guardar a informação das *aliases* corrente.
- *Call Function Graph* (CG) para garantir que não existem *may-aliases*.

## 4.2 Deteção de Vulnerabilidades

Nesta secção iremos ilustrar como é efectuda a deteção de vulnerabilidades, tendo como exemplo o programa PHP da Listagem 4.1. O programa permite verificar se determinado produto especificado pelo utilizador existe e tem desconto. O programa contém uma vulnerabilidade de XSS, no caso do produto não ser encontrado (linha 12).

```
1  <?php
2
3      $produto = $_POST['produto'];
4      if (exiteProduto($produto)) {
5          $produto = htmlentities($produto);
6          if (!temDesconto($produto)) {
7              echo "O produto " . $produto . " não tem desconto";
8          } else {
9              echo "O produto " . $produto . " tem desconto";
10         }
11     } else {
12         echo "O produto " . $produto . " não existe de momento";
13     }
14 ?>
```

Listagem 4.1: Exemplo de programa PHP vulnerável a XSS

O código é inicialmente traduzido para a linguagem intermédia HHAS. A Figura 4.2 apresenta o código da Listagem 4.1 em HHAS sob a forma de um grafo de fluxo de controle (CFG) composto por blocos básicos (BB), numerados de 1 a 8, para facilitar a interpretação.

Este programa exemplifica a pesquisa de um produto que pode ser efectuada por um utilizador. Se considerarmos que as funções `existeProduto` e `temDesconto` podem usar uma variável potencialmente maliciosa, conseguimos observar que, caso o produto exista, é feita a sanitização da variável $produto pela função `htmlentities` no BB 2 (linhas 10, 11 e 12) podendo assim ser utilizada na função *echo* (representada em HHAS por *print*), tal como acontece no BB 4 (linha 24) e BB 5 (linha 33). No caso do produto não existir, não é feita qualquer sanitização. Neste caso, no BB 3 (linha 39) podemos observar que a variável continua a vermelho por manter o seu estado malicioso e, portanto, quando é utilizada pela função *echo* explora a vulnerabilidade XSS.

Para descobrir esta vulnerabilidade e após termos o programa compilado para a linguagem HHAS, o DVHipHop utiliza um *parser* e um *lexer* para criar a AST deste programa. Na Figura 4.3 é ilustrado um exemplo da árvore de *parser* gerada para as duas primeiras instruções (linhas 10 e 11) do BB 2 da Figura 4.2.

Como podemos visualizar na figura, cada *statment* tem a informação da sua localização no código fonte, que irá permitir identificar onde é utilizado um tipo malicioso que torna o programa vulnerável. Deste modo, será possível identificar as

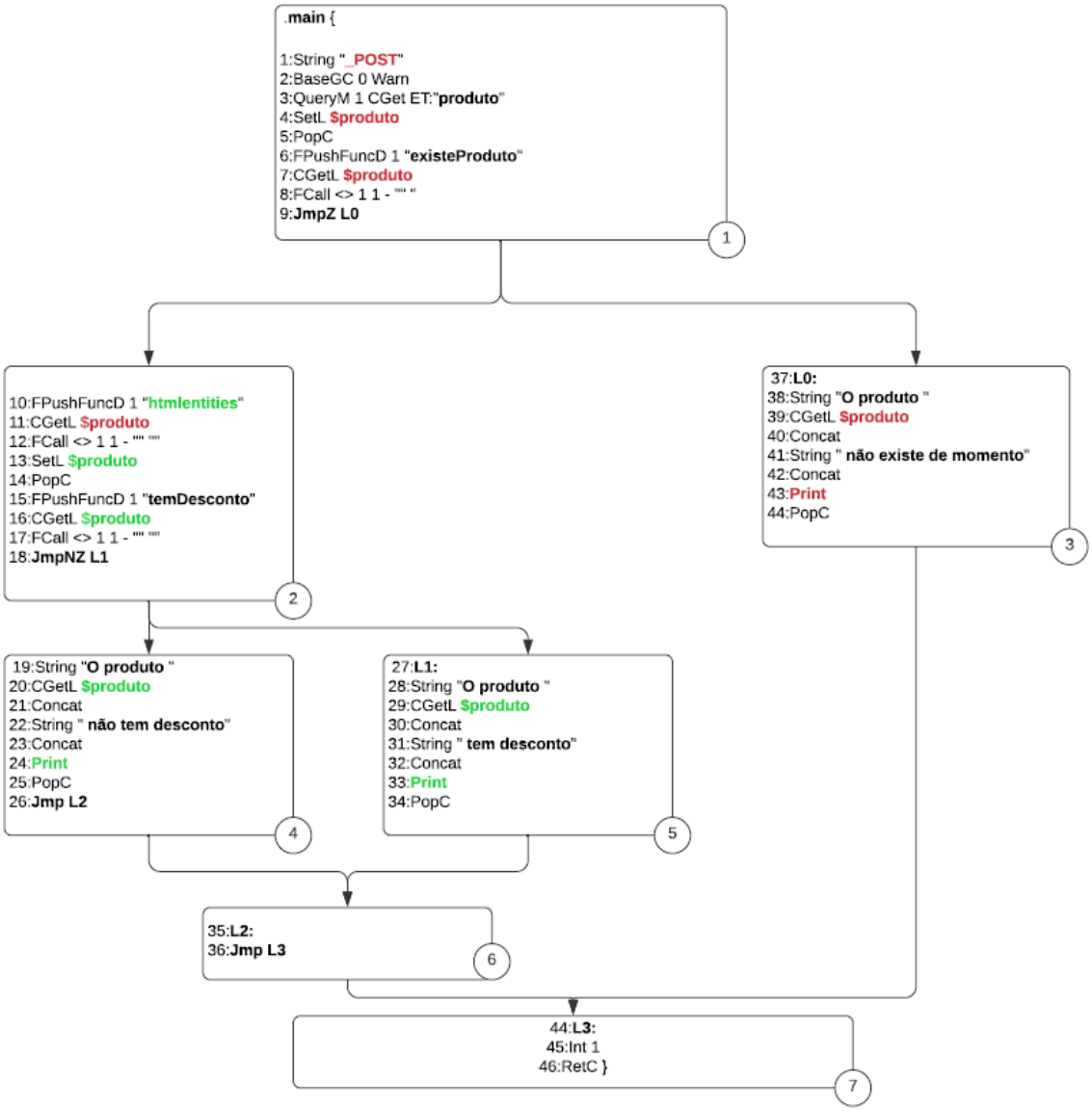


Figura 4.2: Programa da Listagem 4.1 traduzido para HHAS e representado num CFG contendo BBs.

vulnerabilidades no código fonte.

Para analisar esta árvore de *parser* são então feitas as passagens descritas na secção anterior, cada qual efectuada por um visitante que navega ao longo da AST.

As Tabelas 4.2.1-4.2.4 mostram como o programa da Figura 4.2 seria analisado. Cada linha da tabela mostra o estado da análise de tipos depois da instrução da primeira coluna ser considerada. Este estado está representado na tabela da seguinte forma:

- Instrução em análise, apresentada no formato b:i:Programa[i], onde b é o número da *Frame Branch* corrente, i é a linha da instrução a ser analisada e Programa[i] é a instrução. (primeira coluna)

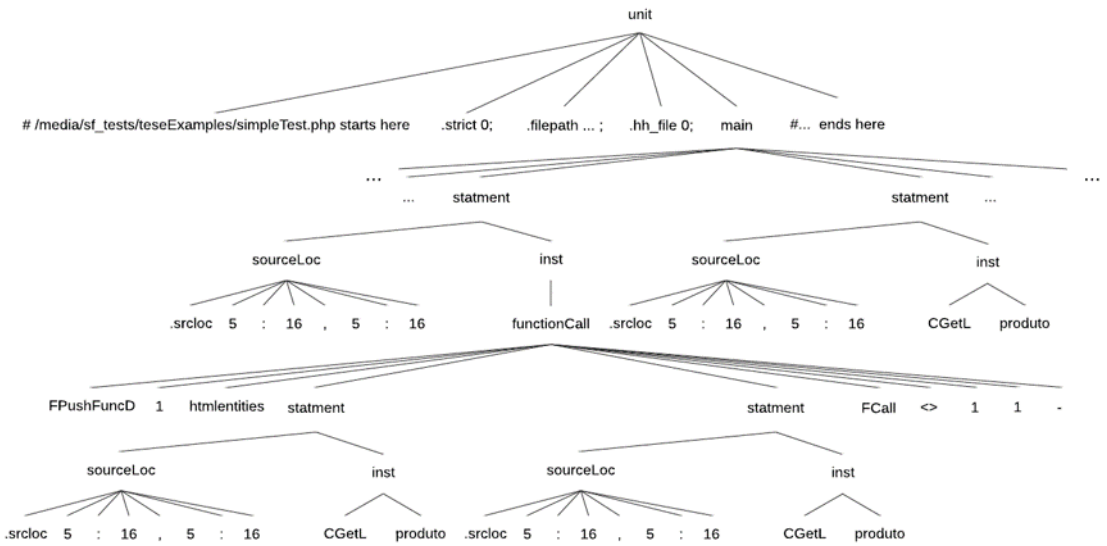


Figura 4.3: Árvore de *Parser* das duas primeiras instruções do bloco básico 2 da Figura 4.2

- Estado da pilha de avaliação, onde cada objeto está definido dentro de um HHTAType que está representado por dois parenteses grandes. (segunda coluna)
- Tabela de símbolos, onde é guardada a VariantBox de cada variável. (terceira coluna)
- Mapa de *Frame Branch*, contém todas as *Frame Branches* da análise mapeadas por uma etiqueta. É apresentado no formato x:y, onde x é o nome da etiqueta e y é o número da *Frame Branch*. (quarta coluna)

Nesta demostração são usadas as seguintes notações:

- *GeneralType* (GT) - representa o tipo geral de dados de uma variável (ex: `Untainted`).
- *PrimitiveGeneralT* (PGT) - é um tipo primitivo representado pelo tipo geral GT
- *VariantBox* (VB) - contém toda a informação de tipos de uma variável. Este objeto representa uma caixa e permite que sejam mantidas mais facilmente as relações de *aliases*.
- *SuperGlobals* (SG) - enumerado das várias super globais consideradas na nossa análise.

- `Untainted` (U) - tipo `Untainted`
- `Tainted` (T) - tipo `Tainted`
- $\epsilon$ - Símbolo que representa o estado vazio. No início da análise o estado inicial das estruturas de dados da segunda, terceira e quarta colunas é vazio, logo representado por $\epsilon$.
- FID - identificador de uma função que contém o nome da função e uma *flag* booleana que indica se a função é a função *main*

Nesta demostração é abstraído: o registo que guarda o apontador para um *array* na instrução número 2 e a pilha de funções prontas a serem chamadas por só ser chamada uma função de cada vez.

Tabela 4.2.1: Demonstração da análise feita pelo analisador DVHipHop ao programa da Figura 4.2

| **Instrução em análise** | **Estado da Pilha de avaliação** | **Tabela de Símbolos** | **Mapa de *Frame Branch*** |
|---|---|---|---|
| 1:1: `String` "*_POST*" | $(\_POST : String)$ | $\epsilon$ | $\epsilon$ |
| 1:2: `BaseGC` 0 *Warn* | $(\_POST : String)$ | $\epsilon$ | $\epsilon$ |
| 1:3: `QueryM1 CGetET`:"produto" | $(((T : GT) : PGT) : VB)$ | $\epsilon$ | $\epsilon$ |
| 1:4: `SetL` $*produto* | $(((T : GT) : PGT) : VB)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\epsilon$ |
| 1:5: `PopC` | $\epsilon$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\epsilon$ |
| 1:6: `FPushFuncD` 1 "*existeProduto*"<br>1:7: CGetL $*produto* | $(((T : GT) : PGT) : VB)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\epsilon$ |
| 1:8: `FCall<>` 1 1 - " " " " | Nota: Verifica se os tipos que estão na pilha são subtipos dos *typehints* definidos na interface da função chamada. Como a função tem como retorno o *typehint* Bool então adiciona `Untainted` ao topo da pilha. | | |
| | $(((U : GT) : PGT) : VB)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\epsilon$ |
| 1:9:`JmpZ` *L0* | Nota: É criada uma cópia do *Frame Branch* atual, a qual é adicionada ao mapa de *Frame Branch*. | | |
| | $\epsilon$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:10: `FPushFuncD` 1 "htmlentities"<br>1:11: CGetL $*produto* | $(((T : GT) : PGT) : VB)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:12: `FCall<>` 1 1 – | Nota: - Como a interface de chamada desta função *built-in* é definida obrigatoriamente no ficheiro de configuração, por se tratar de uma função de sanitização, é adicionado um tipo `Untainted` ao topo da pilha. | | |
| | $(((U : GT) : PGT) : VB)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:13: `SetL` $*produto* | $(((U : GT) : PGT) : VB)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:14: `PopC` | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:15: `FPushFuncD` 1 "*temDesconto*"<br>1:16: `CGetL` $produto | $(((U : GT) : PGT) : VB)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 1:17: `FCall<>` 1 1 – | Nota: Verifica se os tipos que estão na pilha são subtipos dos *typehints* definidos na interface da função chamada. Como a função tem como retorno o *typehint* Bool então adiciona `Untainted` ao topo da pilha. | | |
| | $(((U : GT) : PGT) : VB)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |

Tabela 4.2.2: Demonstração da análise feita pelo analisador DVHipHop ao programa da Figura 4.2 (cont.)

| **Instrução em análise** | **Estado da Pilha** | **Tabela de Símbolos** | **Mapa de *Frame Branch*** |
|---|---|---|---|
| 1:18: `JmpNZ` $L1$ | Nota: É criada uma cópia do *Frame Branch* atual, a qual é adicionada ao mapa B. | | |
| | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:19: `String`"O produto" | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:20: `CGetL` $produto | $\Big(((U : GT) : PGT) : VB)\Big) \cdot \Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:21: `Concat` | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:22: `String`"não tem desconto" | $\Big(((U : GT) : PGT) : VB)\Big) \cdot \Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:23: `Concat` | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:24: `Print` | Nota: Verifica se os tipos que estão na pilha são subtipos dos *typehints* definidos na interface da função chamada. Como a função tem como retorno o *typehint* Bool então adiciona `Untainted` ao topo da pilha. Print não é uma função na linguagem HHAS mas é tratada como tal por ser a operação que representa a função *echo*. | | |
| | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:25: `PopC` | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L1 : 3\}$ |
| 1:26: `Jmp` $L2$ | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ $\{L1 : 3\}$ |
| 3:27: L1 | Nota: Como não existe nenhuma *Frame Branch* no mapa de *Frame Branch* que tenha como chave esta etiqueta desta instrução não é feita nenhuma junção de *Frame Branch*. | | |
| | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:28: `String`"O produto" | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |

Tabela 4.2.3: Demonstração da análise feita pelo analisador DVHipHop ao programa da Figura 4.2 (cont.)

| **Instrução em análise** | **Estado da Pilha de avaliação** | **Tabela de Símbolos** | **Mapa de *Frame Branch*** |
|---|---|---|---|
| 3:29: `CGetL` $*produto* | $\Big(((U : GT) : PGT) : VB)\Big) \cdot \Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:30: `Concat` | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:31: `String`"tem desconto" | $\Big(((U : GT) : PGT) : VB)\Big) \cdot \Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:32: `Concat` | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:33: `Print` | Nota: Verifica se os tipos que estão na pilha são subtipos dos *typehints* definidos na interface da função chamada. Como a função tem como retorno o *typehint* Bool, então adiciona `Untainted` ao topo da pilha. Print não é uma função na linguagem HHAS mas é tratada como tal por ser a operação que representa a função *echo*. | | |
| | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:34: `PopC` | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L2 : 1\}$ |
| 3:35: L2 | Nota: Caso exista alguma *Frame Branch* no mapa de *Frame Branch* que tenha a etiqueta desta instrução então é feita uma junção dos tipos dessa *Frame Branch* com a *Frame Branch* corrente. Neste caso existe a *Frame Branch* 1. Como não existem atribuições a variáveis acedidas dinamicamente em nenhum dos ramos e foi apenas definida a variável $produto que tem o mesmo tipo nos dois ramos, então a *Frame Branch* gerada é igual à *Frame Branch* 1 e 3. | | |
| | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\}$ |
| 3:36: `Jmp` $L2$ | $\epsilon$ | $\{produto : (((U : GT) : PGT) : VB)\}$ | $\{L0 : 2\},\ \{L3 : 3\}$ |
| 2:37: L0 | Nota: Como não existe nenhuma *Frame Branch* no mapa de *Frame Branch* que tenha como chave a etiqueta desta instrução não é feita nenhuma junção de *Frame Branch*. | | |
| | $\epsilon$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |
| 2:38: `String` "*O produto*" | $\Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |

Tabela 4.2.4: Demonstração da análise feita pelo analisador DVHipHop ao programa da Figura 4.2 (cont.)

| **Instrução em análise** | **Estado da Pilha de avaliação** | **Tabela de Símbolos** | **Mapa de *Frame Branch*** |
|---|---|---|---|
| 2:39: `CGetL` $*produto* | $\Big(((T : GT) : PGT) : VB)\Big) \cdot \Big(((U : GT) : PGT) : VB)\Big)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |
| 2:40: `Concat` | $\Big(((T : GT) : PGT) : VB)\Big)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |
| 2:41: `String`"tem desconto" | $\Big(((U : GT) : PGT) : VB)\Big) \cdot \Big(((T : GT) : PGT) : VB)\Big)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |
| 2:42: `Concat` | $\Big(((T : GT) : PGT) : VB)\Big)$ | $\{produto : (((T : GT) : PGT) : VB)\}$ | $\{L3 : 3\}$ |
| 2:42: `Print` | Nota: Verifica se os tipos que estão na pilha são subtipos dos *typehints* definidos na interface da função chamada. Print não é uma função na linguagem HHAS mas é tratada como tal por ser a operação que representa a função *echo*. No ficheiro de configuração que define a interface de funções, para a função *echo* está especificado que é passada uma variável com o tipo `Untainted`. Neste caso é passada uma variável com o tipo `Tainted`, logo é levantada uma exceção TypeMismatchException e o Analisador de Código apresenta a linha onde ocorreu a exceção e identifica a vulnerabilidade de XSS. | | |

## 4.3 Implementação

A solução proposta foi implementada na ferramenta DVHipHop. O componente analisador estático de código foi desenvolvido na linguagem Java que, através de uma programação orientada a objetos, permitiu desenvolver várias estruturas de dados que facilitam a análise estática. O *lexer* e o *parser* para HHAS foram desenvolvidos em Antlr 4 [24]. Nesta secção as principais estruturas de dados descritas na secção anterior são representadas em UML e a implementação dos dois visitantes que fazem as passagens na AST.

A Figura 4.4 apresenta o UML geral da ferramenta DVHipHop. De seguida apresentamos a implementação dos dois visitantes em detalhe, tendo por base este UML.

### 4.3.1 Visitante da Interface das Funções e do *Call Graph*

O primeiro visitante (representado na Figura 4.4 por *FunctionCallInterfaceVisitor*) faz uma passagem pela árvore de *parser* para conhecer todas as funções que vão ser analisadas, de acordo com o descrito na Secção 4.1 para a primeira passagem. Toda a informação recolhida por este visitante guardada no formato representado na Figura 4.5.

Esta figura mostra-nos um diagrama de classes representado em UML, onde não são representados todos os atributos e todas as relações existentes, mas apenas os atributos e as dependências entre classes criados por este primeiro visitante, que se resumem à criação de um *Call Graph* e das várias funções. Nesta primeira fase uma *Function* disponibiliza a seguinte informação:

- A identificação da respetiva função pela classe *FID*, que contém o nome da função e um booleano que indica se a função é *main* ou não.

- A *FunctionInterface* que é a assinatura da função. A sua relação com *Function* é de uma para um, o significa que as chamadas as funções são consideradas monomórficas.

- Um enumerado que distingue as funções definidas pelo utilizador *(BByUser)* das funções *built-in* do PHP, que podem ter a sua interface definida no ficheiro de configuração utilizado pela ferramenta. Estas são representadas pelo enumerado *BInDefined* caso estejam definidas, caso contrário são representadas pelo enumerado *BInUndefined*.

- Uma *FunctionContext* que é a sub-árvore que representa uma função na árvore de *parser*.

Podemos ainda observar que é utilizada a classe *Unit* que na especificação da linguagem HHBC (definida no documento apresentado em [11]) é a unidade que contém toda a informação de um ficheiro PHP ou Hack. Na ferramenta DVHipHop como não são considerados *includes* só existe uma Unit.

### 4.3.2 Visitante da Inferência de Tipos

Este visitante, representado na Figura 4.4 por *TypeInferenceVisitor*, faz a análise de comprometimento baseada em análise de tipos, o que permite aprovar se nenhum tipo malicioso é utilizado numa *sensitive sink*. Caso contrário, este visitante é responsável por levantar uma exceção de *TypeMissMatch* e indicar a zona do código fonte onde o tipo malicioso não foi aceite. Estas exceções são guardadas e caso as assinaturas de todas as funções forem iguais às da análise anterior, o ponto fixo é encontrado e é reportada uma vulnerabilidade, caso contrário a exceção é esquecida e é feita uma nova análise com as novas assinaturas das funções.

O diagrama UML na continuação da Figura 4.4 ajuda a perceber as principais classes, funções e atributos utilizados para fazer esta análise. Este diagrama mostra em mais detalhe as classes usadas para poder efectuar a análise do programa com o visitante *TypeInferenceVisitor*. Nessa análise são usadas várias *TAFrameBranch* que representam uma *Frame* num determinado caminho do programa. Em cada *Frame* é usada uma pilha de avaliação que vai sendo atualizada a cada operação realizada, uma tabela de símbolos para guardar o tipo de cada variável e uma tabela de *aliases*.

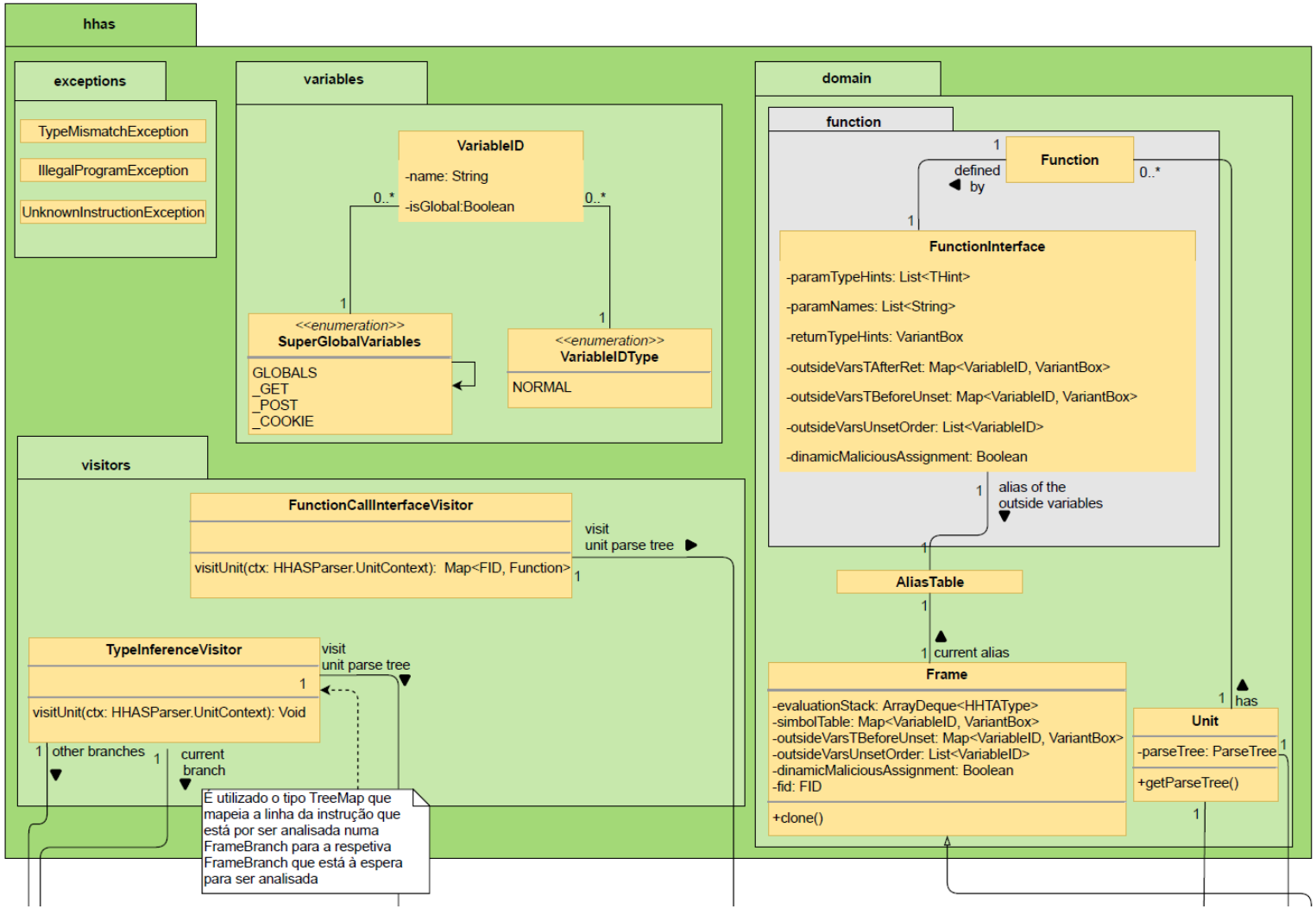


Figura 4.4: Representação UML geral da Ferramenta DVHipHop

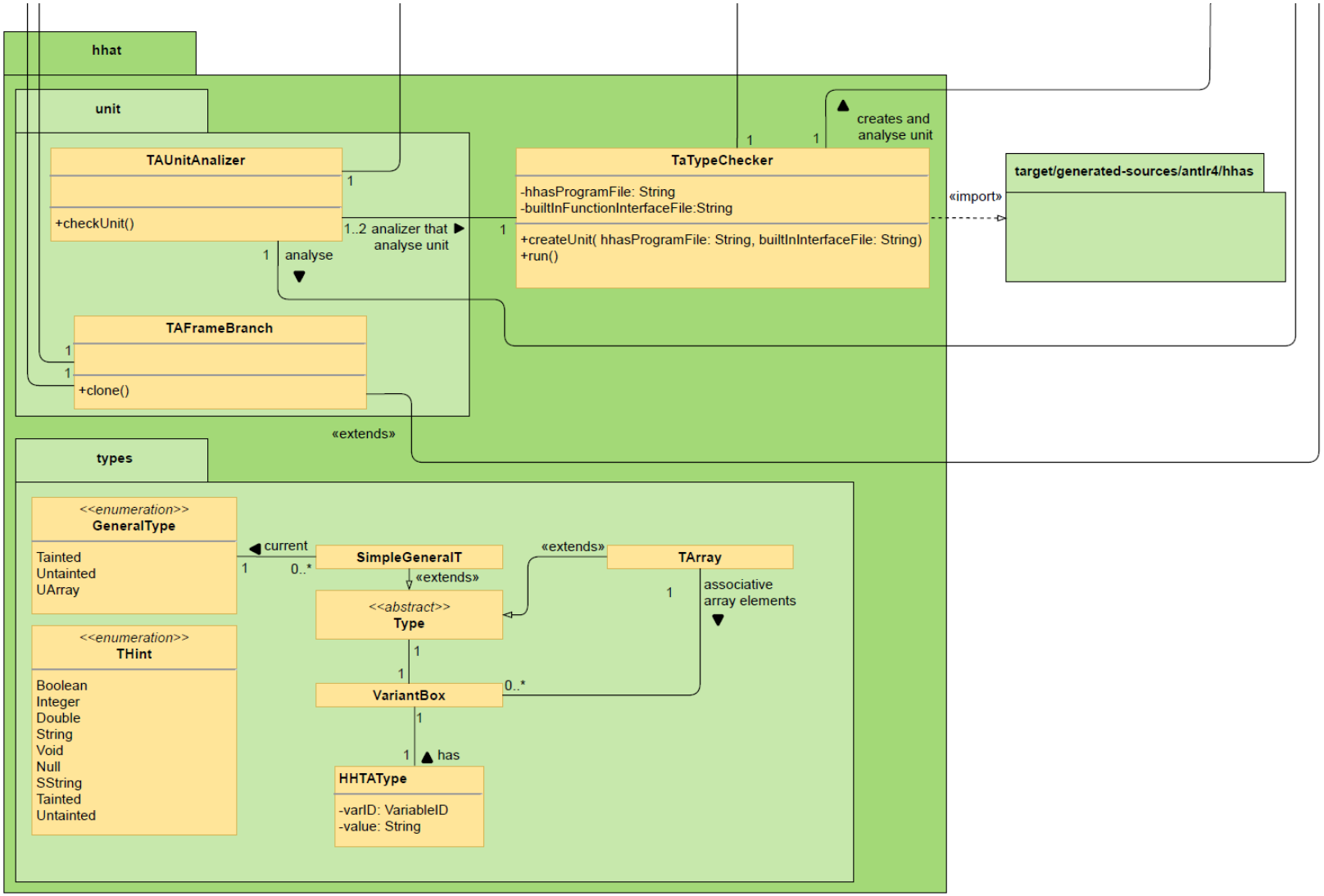


Figura 4.4: Representação UML geral da Ferramenta DVHipHop (continuação)

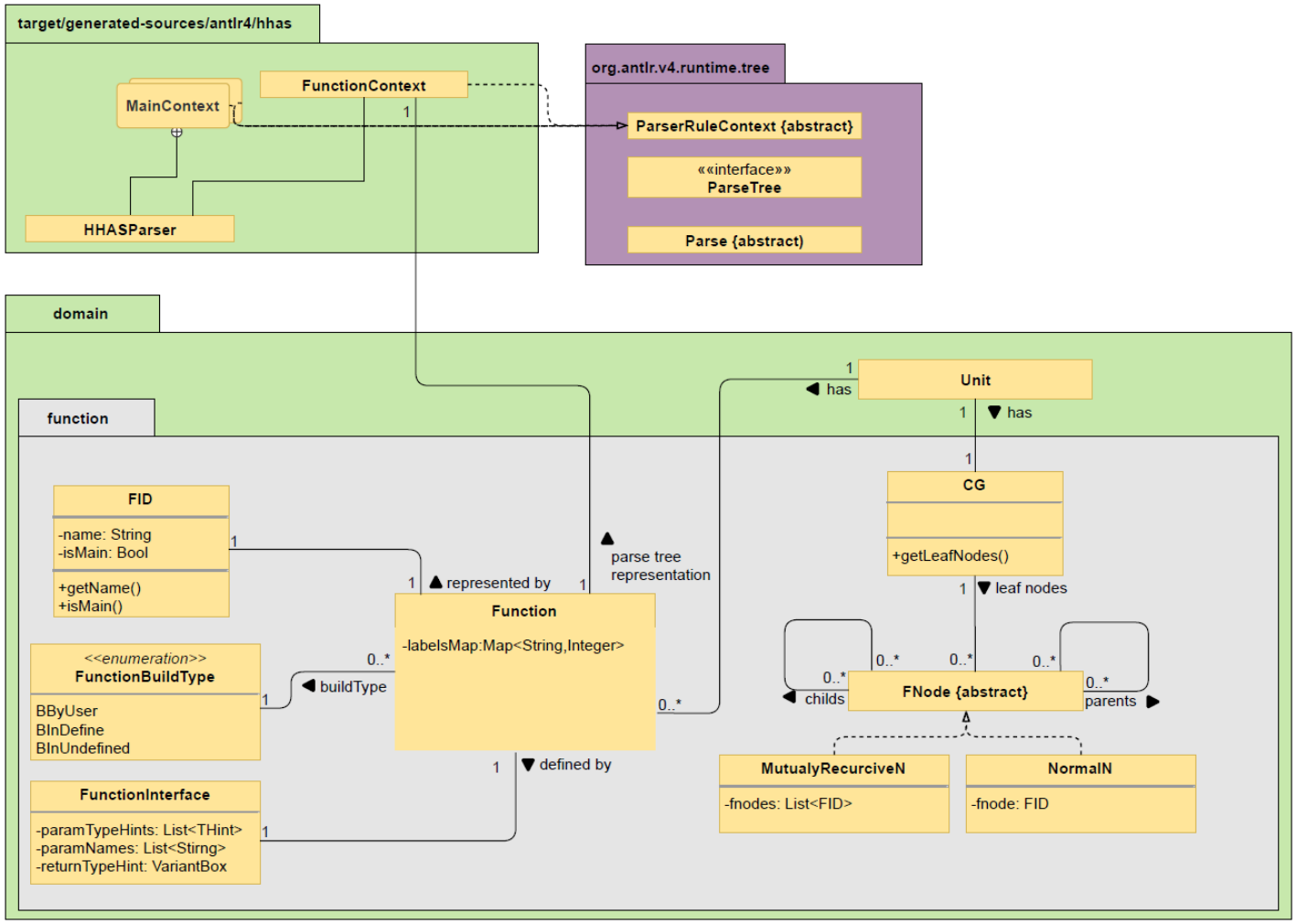


Figura 4.5: Representação UML das Classes existentes depois da passagem do primeiro visitante pela árvore de *parser*

# Capítulo 5

# Avaliação

Neste capítulo são explicadas as técnicas usadas para validar a ferramenta DVHipHop e apresentados os casos de teste criados para mostrar como a ferramenta se comporta em casos tipo. Para além dos testes criados para validar características da ferramenta, foram analisados e corrigidos testes do *Software Assurance Reference Dataset* (SARD) [25, 26], da *National Institute of Standards and Technology* (NIST), para avaliar a ferramenta. O SARD é um repositório de testes que contem excertos de código vulneráveis e não vulneráveis. Assim, através da análise destes excertos foi possível avaliar a capacidade da ferramenta na deteção de vulnerabilidades de XSS e SQLI.

## 5.1 Validação de DVHipHop

Para validar a ferramenta DVHipHop foi criado um sistema de testes com 85 testes JUnit. Cada teste está contido numa categoria para determinada a capacidade da ferramenta para descobrir tipos de variáveis maliciosos e sanitizados em diferentes programas, e consequentemente a descoberta de vulnerabilidades. As categorias são as seguintes e são apresentadas em detalhe de seguida: Testes simples; Junção de *frame branches*; *Call function graph*; Chamada de funções; *Arrays*; *Aliases*; Variáveis acedidas dinamicamente.

O sistema de testes criado, para além de garantir que a ferramenta origina uma exceção na presença de uma vulnerabilidade, consegue ainda garantir que em cada uma das operações intermédias o estado da análise tem o estado esperado. Para conseguirmos observar esses estados intermédios da análise foi criada uma estrutura de dados que denominamos de *Tainted Analysis Type Checking Snap Shooter* (*TATCSnapShooter*). O *TATCSnapShooter* contem um *TATypeChecker* (apresentado na Figura 4.1) que permite fazer a análise dos testes, que ao ser criado utiliza a *framework* mockito[1] para fazer o *spy* do *TypeInferenceVisitior*. Um *spy* é essen-

[1] `https://site.mockito.org/`

cialmente um invólucro numa instância real de um objeto que permite rastrear as interações de chamadas desse objeto para que possam ser verificadas. Este invólucro juntamente com a *class ResultCaptor* que implementa a *class* Answer [2], permitiu-nos fazer uma cópia do estado da análise sempre que o visitante visita um *Statment-Contentex* da árvore de *parser*. O excerto de código seguinte ilustra a criação deste *spy* na primeira linha e a interação de chamada rastreada na segunda linha.

```
1 TypeInferenceVisitor typeInferenceMainV = Mockito
2       .spy(new TypeInferenceVisitor(unit));
3
4 doAnswer(this.resultCaptor).when(typeInferenceMainV)
5                     .visitStatement(any(StatementContext.class));
```

Listagem 5.1: Criação do *TypeInferenceVisitor* através do *spy*

Um exemplo que ilustra a importância de conhecer o estado intermédio da análise são os programas que fazem atribuições de tipos maliciosos a variáveis acedidas dinamicamente (explicado na Secção 3.3.7). Como estas variáveis não são conhecidas estaticamente, todas as variáveis têm de ser consideradas maliciosas a partir desse momento e, portanto, é importante observar isso na tabela de símbolos.

De seguida são apresentadas todas as categorias de testes. Algumas categorias, tal como na Secção 3.3, contêm exemplos de programas PHP. Para cada um destes programas é descrita a sua análise e o estado que é expectável para o teste que valida essa análise. O estado expectável é definido em cada exemplo através do tipo de cada variável e dos tipos contidos nas interfaces das funções, num determinado ponto do programa. O estado expectável é garantido através de afirmações feitas em junit, que garantem que os tipos de cada variável e os tipos contidos nas interfaces das funções correspondem aos tipos contidos na respetiva cópia de *typeInferenceMainV*, num determinado ponto do programa.

### 5.1.1 Testes simples

Os testes simples são maioritariamente compostos por três linhas que testam individualmente cada uma das operações HHAS descritas na Secção 3.1. O programa na Listagem 5.2 apresenta um teste simples onde é expectável que uma exceção *TypeMissMatch* seja levantada após ser chamada a função echo (na linha 5). Esta exceção resulta da impossibilidade de sanitizar a variável de entrada de utilizador `_POST["id_utilizador"]` como é descrito no Capítulo 3 na Secção 3.3.5.

[2] `https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/stubbing/Answer.html`

```
1 <?php
2
3 $_POST['id_utilizador'] = 0;
4 $id_utilizador = $_POST['id_utilizador'];
5 echo $id_utilizador;
6 ?>
```

Listagem 5.2: Teste em PHP, onde é ilustrada uma atribuição a uma variável de entrada de utilizador

### 5.1.2 Junção de *Frame Branches*

Na categoria Junção de *frame branches* os testes têm o objetivo de testar o comportamento do analisador quando é realizada a junção de dois *frame branches* analisados. Tal acontece sempre que existe uma condição ou um ciclo. Neste sentido, foram efectuados vários testes com ciclos e condições que permitem testar a junção de tabelas de símbolos.

```
1  <?php
2
3      function getNumero(string &$porRef):void{
4
5          if($_POST["idUtilizador"] === "admin"){
6              $porRef = 2222;
7          }else{
8              $GLOBALS["aliasesDoExterior2"] = 1111;
9          }
10     }
11 ?>
```

Listagem 5.3: Teste em PHP, onde é ilustrada a junção de dois frame branches

O programa na Listagem 5.3 apresenta uma função que tem de ser analisada com dois *frame branches*, ou seja, os dois ramos (*branches*) da instrução *if*. Quando estes *frames* se juntam, a tabela de símbolos fica com a junção dos dois tipos da variável $porRef (como explicado na Secção 3.3.2 do Capítulo 3). Como o *type hint* do parâmetro `porRef` é *string*, então o seu tipo no inicío da análise da função será `Tainted`. Após ser analisada a condição da linha 5 as duas *frames* antes de se juntarem, guardam a variável `porRef` na tabela de símbolos com tipos diferentes, ou seja, no *then* o tipo guardado é `Untainted` e no *else* este é `Tainted`. Portanto, após ser feita a junção dos *frame branches* essa variável passa a ter o tipo `Tainted`. Como a variável `GLOBALS["aliasDoExterior2"]` tem o tipo `Untainted` na condição *else* e não é definida no *then* irá ser feita a junção dos tipos `Untainted` e `Tainted` (definido no na Secção 3) e por isso passa a ter o tipo `Tainted` após a junção.

Um outro exemplo de programa que permite fazer um teste desta categoria é o apresentado na Listagem 5.4. Neste programa é inicialmente criado o array seguro `b` e dentro do `if` são feitas atribuições a posições do *array* `a` que não foi criado de uma forma segura. Para este programa foi criado um teste JUnit que verifica se à

saída da instrução `if` a variável `b` tem o tipo `Untainted TArray`, e a varável `a` tem o tipo `Tainted TArray` com todas as posições `Tainted`.

```
1  <?php
2
3  function limpaArray() {
4    $b = array();
5      if ($_POST["user_id"] === "admin") {
6      $a[2] = "Untainted";
7      } else {
8      $a[3] = "Untainted";
9      $b[0] = "Untainted";
10     $b[1] = "Untainted";
11     }
12
13   return $b;
14 }
15 ?>
```

Listagem 5.4: Segundo teste com junção de dois *frame branches*

O teste da Listagem 5.3 faz a junção dos tipos primitivos `Tainted` e `Untainted` da mesma variável, enquanto o teste da Listagem 5.4 faz a junção de *arrays*. A junção de *frame branches* é explicada no Capítulo 3.3.2.

### 5.1.3 Chamada de Funções

Na categoria Chamada de Funções são criados testes com o objetivo de analisar o corpo da função e criar a respetiva interface de chamada da função. Isto permite validar se a interface criada é a *interface* esperada.

Por exemplo, a análise do programa da Listagem 5.5 é feita com duas passagens pelo programa. Após a última passagem é validada a assinatura de cada uma das funções. Para a função `f2` é esperado que tenha definido na sua assinatura as variáveis globais `idUtilizador1`, `idUtilizador2` e `idUtilizador3` com o tipo `Untainted`. Na função `f1` são feitas 2 atribuições a variáveis globais. Na primeira atribuição todas as variáveis passam a ser do tipo `Tainted` por ser feita uma atribuição de uma entrada de utilizador a uma variável global. Na linha 8 a variável `idUtilizador2` fica com o tipo `Untainted` por ser feita uma atribuição de uma *String* segura. Para além disso, é feita uma chamada à função `f2` dentro de uma condição. A chamada a esta função irá alterar o tipo das três variáveis globais, definidas na sua interface, para `Untainted`. Após ser analisada esta condição, é necessária a junção da tabela se símbolos das duas *Frame Branches*. Após essa junção, a análise da função termina e é criada a interface de chamada da função `f1`. O teste valida este programa exigindo que a interface da função `f1` tenha a variável `idUtilizador2` com o tipo `Untainted` e as variáveis `idUtilizador1` e `idUtilizador3` com o tipo `Tainted`.

```php
<?php

  f1();
  echo $GLOBALS["idUtilizador2"];

  function f1():void{
    $GLOBALS["idUtilizador1"] = $_POST["idUtilizador"];
    $GLOBALS["idUtilizador2"] = "sstring";
    if(rand(-2, 2) > 0){
      f2($i);
    }
  }

  function f2():void{
    $GLOBALS["idUtilizador1"] = "sstring";
    $GLOBALS["idUtilizador2"] = "sstring";
    $GLOBALS["idUtilizador3"] = "sstring";
  }
?>
```

Listagem 5.5: Teste de chamadas de funções

### 5.1.4 *Arrays*

Os testes da categoria *Arrays* têm sobretudo em conta dois tipos de operações. Por um lado, as atribuições e o acesso a posições desconhecidas de *arrays* e, por outro, as atribuições e o acesso de posições conhecidas. A Listagem 5.6 apresenta um exemplo de um teste que faz atribuições de tipos maliciosos a posições desconhecidas, verificando assim que todas as posições do *array datas* devem ser `Tainted` exceto a posição com a chave `Miguel` que deve ser `Untainted` por ter sido feita uma atribuição a esse elemento do *array* posterior a todas as atribuições desconhecidas.

```php
<?php

  $mapaDatas = array();
  $mapaDatas["Margarida"] = "1997";
  $mapaDatas["Cristina"] = "1964";
  $nUtilizadoresRandom = 4;

    for ($i = 0; $i < nUtilizadoresRandom; $i++) {
        $mapaDatas[$_POST["nome"]] = $_POST["idade"];
    }

    $mapaDatas["Miguel"] = "1964";
    echo "O Miguel nasceu em ".$datas["Miguel"];
?>
```

Listagem 5.6: Teste com *arrays*

### 5.1.5 *Aliases*

Na categoria *Aliases* foram feitos vários testes para analisar a tabela de *aliases* de programas que fazem *aliases* entre variáveis locais e variáveis globais, duas variáveis locais e duas variáveis globais. Estas relações de aliases são testadas não só no contexto main como dentro de uma função, o que permite perceber se as *aliases* entre variáveis do exterior são mantidas após a chamada a uma função. Também, foram criados testes que permitem verificar se é originada uma exceção de *IllegalProgramException* caso sejam analisados programas com *may-aliases*. A Listagem 5.7 mostra um programa que usa *may-aliases* e, portanto, é expectável que uma exceção *IllegalProgramException* seja levantada após ser feita a análise da chamada à função na linha 8.

```
1  <?php
2
3      $n = $_POST["numeroEscolhido"];
4
5      switch ($n) {
6          case "123":
7              echo "Vitoria!";
8              aliasesGlobais();
9              break;
10         case "111":
11             echo "Segundo lugar!";
12             break;
13         default:
14             echo "Número incorreto.";
15     }
16
17     function aliasesGlobais():void{
18         $GLOBALS["var1"] = &$GLOBALS["var2"];
19     }
20 ?>
```

Listagem 5.7: Teste com *aliases* de variáveis

### 5.1.6 Variáveis acedidas dinamicamente

Os testes feitos na categoria Variáveis acedidas dinamicamente são feitos sobre programas onde existem atribuições e acessos a variáveis de forma dinâmica, para além de ser verificado o caso mais comum em que é feita uma atribuição maliciosa a uma variável variável, também são testados casos particulares, como por exemplo fazer uma atribuição de um tipo malicioso a uma posição desconhecida de um *array* acedido através de uma variável com nome variável. A ferramenta passou em todos os testes desta categoria com êxito.

## 5.2 Avaliação com o SARD

Para além dos testes JUnit desenvolvidos especificamente para a ferramenta DVHipHop, foram também usados testes do SARD, que é um *dataset* de testes geridos e mantidos pelo NIST. Deste *dataset* foram usados apenas um subconjunto dos testes escritos em PHP, nomeadamente 8224 testes, dos quais 250 são vulneráveis. Os testes no SARD são organizados por categorias CWE[3] (Common Weakness Enumeration) que identificam a possível vulnerabilidade contida neles. O subconjunto de testes usado foram das categorias CWE_79 e CWE_98 representantes das vulnerabilidades de XSS e SQLI, respetivamente. Neste subconjunto estão todos os testes considerados bons pelo SARD (que não são vulneráveis) e que não têm como entrada de utilizador a leitura de um ficheiro. Dos testes considerados maus pelo SARD (que são vulneráveis) foram apenas usados os testes que não usam nenhuma função de sanitização. Os testes que usam indevidamente uma função de sanitização não foram utilizados porque representam casos em específico que não são considerados pela ferramenta DVHipHop por não ser feita uma análise de constantes. No total, o subconjunto de testes considerados contém 3222 testes de XSS bons, 250 testes de XSS maus e 4752 testes de SQLI bons, que podem ser encontrados em [25, 27, 26], respetivamente. Cada teste consiste num ficheiro composto por três secções:

- **Entrada de Utilizador**: Nesta secção é feita uma atribuição de uma entrada de utilizador a uma variável local `$tainted`. Esta entrada de utilizador pode ser: o valor de uma variável superglobal, a execução de um comando *shell*, a execução de um programa externo ou a leitura de um ficheiro.

- **Sanitização**: Nesta secção é realizada a sanitização do valor malicioso obtido através da entrada de utilizador. Em alguns casos esta secção pode ser opcional, mas os testes que consideramos têm todos sanitização. O método de sanitização pode ser através de: uma função de sanitização específica da linguagem, a utilização de um *cast*, a inferência de tipos, ou a utilização de uma condição trenaria. Esta secção é opcional e nos testes que consideramos apenas está presente nos testes considerados bons pelo SARD.

- ***Sensitive Sink***: Nesta secção é utilizada uma função da linguagem que é "sensível" aos valores dos parâmetros que lhe são passados, podendo originar resultados inesperados. Assim, a uma *sensitive sink* pode ser passado como parâmetro uma variável que pode ter um valor malicioso.

[3] `https://cwe.mitre.org/`

```
1  <!DOCTYPE html>
2  <html>
3  <head/>
4  <body>
5  <?php
6  $tainted = $_GET['UserData'];
7
8  $tainted = htmlentities($tainted, ENT_QUOTES);
9
10
11 echo "<div id='".  $tainted ."'>content</div>" ;
12 ?>
13 <h1>Hello World!</h1>
14 </body>
15 </html>
```

Listagem 5.8: Teste SARD 191440, não vulnerável a XSS

O programa da Listagem 5.8 é um exemplo de um teste do SARD. Neste programa a linha 6 corresponde à secção Entrada de Utilizador, a linha 8 corresponde à secção de Sanitização e a linha 11 à *Sensitive Sink*.

Os nomes dos testes SARD para PHP são organizados da seguinte forma: CWE <CWE ID> <entrada de utilizador> <método de sanitização> <localização onde o valor, possivelmente malicioso, é utilizado>.php Isto permitiu decidir em que testes faria sentido testar a ferramenta DVHipHop e de que forma estariam organizados.

Como na análise de tipos descrita na Secção 3.3 só é considerado um tipo de `Untainted` que é sanitizado para todas as funções *sensitive sink* (tanto a função *mysql_query* como *echo* têm como *type hint* o tipo `Untainted`), muitos dos testes SARD analisados são semelhantes. Por exemplo, os testes representados nas Listagens 5.8 e 5.9 podem ser considerados semelhantes, pois só são alteradas as `Sstring` concatenadas à variável `$tainted` que são consideradas pela ferramenta DVHipHop por não considerar análise de constantes.

As entradas de utilizador consideradas nestes testes podem ser divididas em três grupos representados na Tabela 5.2.1

Tabela 5.2.1: Entradas de utilizador contidas nos testes SARD considerados

| | | |
|---|---|---|
| **Entradas de Utilizador** | Superglobal diretamente | $_GET |
| | | $_POST |
| | | $_SESSION |
| | Superglobal através de um array | *array*[] =_GET['userData'] ; |
| **Funções** | Através da execução de *script* | 'cat /tmp/tainted.txt' |
| | | exec($script, $result, $return); |
| | | shell_exec('cat /tmp/tainted.txt') |
| | | system('ls', $retval) |
| | Através da desserialização de uma entrada de utilizador | $string = $_POST['UserData'] ; unserialize($string); |

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <script>
5  <?php
6  $tainted = $_GET['UserData'];
7
8  $tainted = htmlentities($tainted, ENT_QUOTES);
9
10
11 echo "alert('". $tainted ."')" ;
12 ?>
13 </script>
14 </head>
15 <body>
16 <h1>Hello World!</h1>
17 </body>
18 </html>
```

Listagem 5.9: Teste SARD 192442, idêntico ao teste da listagem 5.8

Se dois testes tiverem a mesma função de sanitização e tiverem como entrada de utilizador duas entradas que pertençam ao mesmo grupo de entradas de utilizador então esses testes são considerados semelhantes pela nossa análise.

Para além de podermos agrupar testes semelhantes consoante a entrada de utilizador, também, é possível fazer o mesmo com as funções de sanitização. A Tabela 5.2.2 mostra os vários grupos de diferentes métodos de sanitização utilizados nos testes considerados.

Tabela 5.2.2: Métodos de sanitização contidos nos testes do SARD considerados

| **Categoria de sanitização** | **Método** |
|---|---|
| Sem Sanitização | |
| Inferência de tipo através da operação HHAS CastInt ou CastDouble | Cast float |
| | Cast int |
| Inferência de tipo através da operação HHAS SetOpL | Cast através de += 0 |
| | Cast através de += 0.0 |
| Inferência de tipo através da operação HHAS Add | Cast através de + 0 |
| Condição ternária | $tainted = $tainted == 'safe1' ? 'safe1' : 'safe2'; |
| Função que não depende de filtro ou expressão regular | rawurlencode |
| | addslashes |
| | urlencode |
| | http_build_query |
| | htmlentities |
| | htmlspecialchars |
| | mysql_real_escape_string |
| | intval |
| | floatval |
| Função que depende de filtro ou expressão regular | filter_var |
| | preg_replace |
| Inferência de tipo através do predicado de uma condição | if(settype($tainted,"integer")) |
| | $legal_table = array("safe1", "safe2"); if (in_array($tainted, $legal_table, true)) |

As funções `filter_var` e `preg_replace` estão agrupadas na Tabela 5.2.2 por dependerem de um filtro ou expressão regular específico para determinada situação. Esta característica não permite que a ferramenta DVHipHop consiga determinar, com a sua análise, se a função faz ou não a sanitização. Para que isso fosse possível era necessário fazer uma análise de constantes, que permitisse saber qual o filtro passado como parâmetro. Para além disso, era necessário considerar funções polimórficas, onde o tipo de retorno iria depender do tipo dos parâmetros passados. Se estas funções fossem definidas no ficheiro de configuração de modo a retornarem uma `Sstring` iriamos obter o resultado esperado nos testes que consideramos, mas estaríamos por outro lado a admitir que no caso de não ser passado o filtro correto a ferramenta iria gerar falsos negativos e, portanto, foi definido que estas funções não fazem a sanitização.

O método de sanitização que utiliza inferência de um tipo através do predicado de uma condição também não foi considerado na análise de tipos. Para além disso, este método que utiliza funções como o `settype` iria também exigir a análise de constantes para apurar o tipo passado no segundo parâmetro.

Na secção das *sensitive sinks* é utilizada uma *sensitive sink* diferente em cada uma das categorias de vulnerabilidades CWE, onde é testada a *sensitive sink* `echo` nos testes de XSS e a função *mysql_query* nos testes de SQLI. Apesar de só serem testadas duas funções de *sensitive sink*, muitos testes diferenciam-se apenas por pequenas alterações feitas nesta secção que alteram o parâmetro que é passado pela *sensitive sink*. Podemos observar uma destas alterações entre os programas das Listagens 5.8 e 5.9. Como a ferramenta DVHipHop não faz análise de constantes, todos os testes que se diferenciam apenas nesta secção são considerados iguais.

Dos 3472 casos de XSS analisados, a ferramenta reportou corretamente 2366 casos ao fazer a análise de testes bons sem levantar uma exceção e 250 casos ao fazer a análise de testes maus levantando uma exceção de *TypeMissMatch*. Os restantes 856 casos resultaram em falsos positivos, ou seja, o DVHipHop levantou indevidamente uma exceção *TypeMissMatch*. Os falsos positivos estão associados com os testes contidos nos últimos dois grupos da Tabela 5.2.2, e consequentemente pelos motivos explicados acima.

Para SQLI, dos 4752 testes, foram corretamente classificados somente 1386, enquanto os restantes 3566 casos foram falsos positivos. Esta grande discrepância entre resultados explica-se sobretudo na forma como a *query*, onde é gerada para ser passada como parâmetro à *sensitive sink* `mysql_query`, é utilizada a função `sprintf` que tem o seu tipo de retorno definido no ficheiro de configuração como sendo do tipo `String` malicioso (definido na Secção 3.3). Tal como foi explicado anteriormente, este problema só seria resolvido com funções polimórficas e análise de constantes. O teste apresentado na Listagem 5.10 mostra o exemplo de um destes casos. Para

além disso, tal como nos testes de XSS, são obtidos alguns falsos positivos por não serem utilizados os dois últimos grupos de métodos de sanitização da Tabela 5.2.2.

```
1  <?php
2
3  $tainted = $_GET['UserData'];
4
5  $tainted = mysql_real_escape_string($tainted);
6
7  $query = sprintf("SELECT * FROM student where id=%d", $tainted);
8
9  $conn = mysql_connect('localhost', 'mysql_user',
       'mysql_password'); // Connection to the database (address,
       user, password)
10 mysql_select_db('dbname') ;
11 echo "query : ". $query ."<br /><br />" ;
12
13 $res = mysql_query($query); //execution
14
15 while($data =mysql_fetch_array($res)){
16 print_r($data) ;
17 echo "<br />" ;
18 }
19 mysql_close($conn);
20
21 ?>
```

Listagem 5.10: Teste SARD 163812, não vulnerável a SQLI

Nestes resultados podemos observar que a ferramenta é capaz de analisar uma grande quantidade de testes corretamente e que, apesar de existirem alguns falsos positivos, a ferramenta não gera nenhum falso negativo.

# Capítulo 6

# Conclusão e Trabalho Futuro

Neste capítulo são apresentadas as conclusões do trabalho e são expostas algumas ideias de trabalho futuro que permitiriam melhorar a ferramenta.

## 6.1 Conclusão

Este trabalho de tese teve como objetivo criar uma ferramenta capaz de analisar estaticamente programas compilados para uma linguagem intermédia que tenham entradas de utilizador por onde podem ser passados valores maliciosos, os quais se usados em *sensitive sinks* os tornam vulneráveis a ataques de *SQLI* e *XSS*.

Para o efeito, foi inicialmente efectuado um estudo sobre estas classes de vulnerabilidades e uma revisão do estado de arte dos mecanismos que têm sido desenvolvidos para colmatar este problema, tal como a utilização de analisadores de tipos e a utilização de *machine learning*. A partir destes conhecimentos foi possível desenvolver uma ferramenta, denominada DVHipHop, que através da análise de tipos consegue descobrir vulnerabilidades de XSS e SQLI na linguagem compilada e identificá-las no código fonte. Para que a DVHipHop fosse capaz de analisar vários programas escritos em linguagens de programação PHP e Hack foi utilizada uma linguagem de baixo nível. A ferramenta é composta por dois componentes onde, o primeiro traduz um programa para a linguagem HHAS, e depois usa um *lexer* e um *parser* para criar a *AST*, e o segundo utiliza vários visitantes que efectuam passagens pela *AST* para realizar a análise de tipos e de comprometimento com objectivo de detectar vulnerabilidades.

A ferramenta foi validada por um sistema de testes criado para o efeito, onde foram criados vários testes para cada funcionalidade da ferramenta, e avaliada com um conjunto de testes do *SARD*. Estes testes mostraram que a ferramenta é capaz de encontrar vulnerabilidades, tanto em programas simples, como em programas com funcionalidades dinâmicas. No sistema de testes todos os testes deram o resultado esperado. Nos testes SARD os resultados mostraram que a ferramenta é capaz de

validar uma grande quantidade de testes que fazem devidamente a sanitização de uma variável que tem um tipo malicioso, mas que por outro lado mostraram que a ferramenta também gera falsos positivos por não recorrer a análise de constantes e ao polimorfismo.

Deste trabalho pode-se concluir que a análise estática, efectuada sobre uma linguagem intermédia e de baixo nível, permite analisar todas as linguagens que possam ser compiladas para essa linguagem. Para além disso, pelo facto da análise estática ser feita sobre uma linguagem dinâmica podemos observar que quando são consideradas as suas funcionalidades dinâmicas, como por exemplo os tipos dinâmicos e as variáveis acedidas dinamicamente, não é possível garantir estaticamente todos os estados do programa possíveis em tempo de execução, e que portanto, como esses estados não são conhecidos é necessário considerar o pior caso possível, para garantir que a análise não gera falsos negativos.

## 6.2 Trabalho Futuro

Dada a importância dos rápidos ciclos de desenvolvimento que são privilegiados por linguagens como o PHP e o Hack, seria interessante fazer com que a ferramenta mantivesse esta característica, fazendo a análise em paralelo de todas as funções, tal como a análise em paralelo de todas as *FrameBranches* pertencentes à análise de uma função.

A ferramenta desenvolvida faz a análise da linguagem HHAS mas não suporta todas as suas funcionalidades, como por exemplo, os objetos e os *includes*. Apesar destas funcionalidades não terem sido implementadas foram tidas em consideração durante a criação da ferramenta, ao assumir que qualquer *array* poderia ser um objeto e que seria possível implementar os *includes* se fosse usada a mesma abordagem utilizada nas funções e nos ciclos, onde é usado um algoritmo de calculo de ponto fixo, que permitisse analisar as *Units* de forma individual.

Neste momento está a ser considerado que qualquer função de sanitização torna um valor `Untainted` para qualquer *sensitive sink*. A ferramenta podia ser melhorada se fossem criados vários subtipos de `Untainted` que permitissem definir que uma função de sanitização torna um valor `Untainted` apenas para determinadas *sensitive sinks*.

Fica ainda por ser criado um teorema de type soundness que garanta que programas bem tipificados são bem-comportados.

# Bibliografia


[1] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (Technical report). *Secure Systems Lab, Vienna University of Technology*, 2006.

[2] Ibéria Medeiros, Nuno Neves, and Miguel Correia. DEKANT: A Static Analysis Tool That Learns to Detect Web Application Vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2016, pages 1–11, Saarbrücken, German, July 2016. ACM.

[3] W3Techs Web Technology Surveys. https://w3techs.com/technologies/overview/programming_language/.

[4] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop Compiler for PHP. *In Proceeding of the ACM SIGPLAN Notices*, 47(10):575–586, October 2012.

[5] Andrei Homescu and Alex Şuhan. HappyJIT: A Tracing JIT Compiler for PHP. In *Proceedings of the Symposium on Dynamic Languages*, DLS '11, page 25–36, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. Myths and facts about static application security testing tools: An action research at telenor digital. In *Proceeding of the International Conference on Agile Software Development*, pages 86–103. Springer, 2018.

[7] Zixiang Zhu. Automated Penetration Testing for PHP Web Applications. Master's thesis, Georgia Institute of Technology, 2017.

[8] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *In Proceeding of the IEEE Transactions on Reliability*, 65(1):54–69, 2016.

[9] Jeesoo Jurn, Taeeun Kim, and Hwankuk Kim. An Automated Vulnerability Detection and Remediation Method for Software Security. *In Proceeding of the Sustainability (ISSN 2071-1050; CODEN: SUSTDE)*, 10(5):1652, 2018.

[10] CVE Common Vulnerability Enumeration. `https://www.cvedetails.com`.

[11] HHBC bytecode specification. `https://github.com/facebook/hhvm/blob/master/hphp/doc/bytecode.specification`.

[12] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *Proceedings of the International Acm Sigplan Notices*, volume 49, pages 777–790. ACM, 2014.

[13] J. Williams and D. Wichers. OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks, 2017.

[14] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, pages 199–209. IEEE Computer Society, 2009.

[15] SQL INJECTION cheat sheet & tutorial: Vulnerabilities & how to prevent sql injection attacks. `https://www.veracode.com/security/sql-injection`.

[16] Stephanos Mavromoustakos, Aakash Patel, Kinjal Chaudhary, Parth Chokshi, and Shaili Patel. Causes and Prevention of SQL Injection Attacks in Web Applications. In *Proceedings of the International Conference on Information and Network Security*, pages 55–59. ACM, 2016.

[17] Ibéria Medeiros, Miguel Beatriz, Nuno Neves, and Miguel Correia. SEPTIC: Detecting Injection Attacks and Vulnerabilities Inside the DBMS. *IEEE Transactions on Reliability*, 68(3):1168–1188, Sept 2019.

[18] Guilherme Ottoni. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165. ACM, 2018.

[19] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[20] JJ Kronjee. Discovering vulnerabilities using data-flow analysis and machine learning. Master's thesis, Open Universiteit Nederland, 2018.

[21] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006.

[22] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceeding of the ACM Sigplan Notices*, volume 45, pages 377–388. ACM, 2010.

[23] Stephen N Freund and John C Mitchell. A type system for the java bytecode language and verifier. *In Proceeding of the Journal of Automated Reasoning*, 30(3-4):271–321, 2003.

[24] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[25] Bertrand Stivalet and Aurelien Delaitre. Testes escritos em php da categoria cwe_79 que são considerados bons pelo sard. `https://samate.nist.gov/SARD/view.php?fromWhere=fromSearch&reference=189152-199217&description=&author=&flawed%5B%5D=Good&languages%5B%5D=PHP&typesofartifacts%5B%5D=Source+Code&status_Candidate=1&status_Accepted=1&weakness_exact=false&flaw=Any...&fileName=&minFileSize=&maxFileSize=&minFiles=&maxFiles=&date=&typeDate=Any...&Submit=Search+Test+Cases`.

[26] Bertrand Stivalet and Aurelien Delaitre. Testes escritos em php da categoria cwe_98 que são considerados bons pelo sard. `https://samate.nist.gov/SARD/view.php?count=20&description=Safe+sample&fileName=CWE_89.%2A%3F%5C.php&searchNameByRegex=1&languages[]=PHP&flawed[]=Good&status_Candidate=1&status_Accepted=1&first=0&sort=desc`.

[27] Bertrand Stivalet and Aurelien Delaitre. Testes escritos em php da categoria cwe_79 que são considerados maus pelo sard. `https://samate.nist.gov/SARD/view.php?fromWhere=fromSearch&reference=&description=&author=&flawed%5B%5D=Any...&languages%5B%5D=PHP&typesofartifacts%5B%5D=Source+Code&status_Candidate=1&status_Accepted=1&weakness_exact=CWE-089%3A+Improper+Neutralization+of+Special+Elements+used+in+an+SQL+Command+%28%27SQL+Injection%27%29&flaw=CWE-089%3A+Improper+Neutralization+of+Special+Elements+used+in+an+SQL+Command+%28%27SQL+Injection%27%29&fileName=&minFileSize=&maxFileSize=&minFiles=&maxFiles=&date=&typeDate=Any...&Submit=Search+Test+Cases`.

[28] Supeno Djanali, FX Arunanto, Baskoro Adi Pratomo, Hudan Studiawan, and Satrio Gita Nugraha. SQL injection detection and prevention system with raspberry Pi honeypot cluster for trapping attacker. In *Proceeding of the Techno-*

*logy Management and Emerging Technologies, 2014 International Symposium on*, pages 163–166. IEEE, 2014.

[29] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceeding of the Network and Distributed System Security Symposium*, pages 2000–02, 2000.

[30] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *In Proceeding of the IEEE Software*, 19(1):42–51, 2002.